

# **Verification & Validation Of Neural Networks For Aerospace Systems**



COLLABORATION BETWEEN  
Dryden Flight Research Center  
AND  
NASA Ames Research Center

Dated: June 12, 2002  
Contributors: Dale Mackall, DFRC  
Stacy Nelson, NCC  
Johann Schumman, RIACS

Special thanks to the individuals listed below (in alphabetical order by first name) for their time in answering questions and providing project/technical expertise:

Brian Taylor, ISR  
Chuck Jorgensen, ARC  
Dick Larson, DFRC  
Don Solloway, ARC  
Greg Limes, ARC  
John Burken, DFRC  
Karen Gundy-Burlet, ARC  
Kurt Guenther, DFRC  
Mark Boyd, ARC  
Pamela Parker, NCC  
Peggy Williams, DFRC  
Timothy Smith, Boeing  
Van Casdorph, ISR

ARC – Ames Research Center  
DFRC – Dryden Flight Research Center  
ISR – Institute for Software Research, Inc.  
NCC – Nelson Consulting Company  
RIACS - Research Institute for Advanced Computer Science

## TABLE OF CONTENTS

1.	<a href="#">NOMENCLATURE</a>	5
2.	<a href="#">EXECUTIVE SUMMARY</a>	6
2.1.1.	<a href="#">OVERVIEW OF ADAPTIVE SYSTEMS</a>	6
2.1.2.	<a href="#">V&amp;V PROCESSES/METHODS</a>	6
3.	<a href="#">OVERVIEW OF ADAPTIVE SYSTEMS</a>	10
3.1.	<a href="#">Overview of Neural Networks</a>	10
3.2.	<a href="#">Pre-Trained Neural Networks (PTNN)</a>	12
3.3.	<a href="#">Online Learning Neural Networks (OLNN)</a>	14
4.	<a href="#">V&amp;V PROCESSES/METHODS</a>	16
4.1.	<a href="#">Overview of NASA V&amp;V Standards</a>	16
4.1.1.	<a href="#">Life Cycle</a>	17
4.2.	<a href="#">Verification of Adaptive Systems</a>	18
4.2.1.	<a href="#">Contract Verification</a>	18
4.2.2.	<a href="#">Process Verification</a>	19
4.2.3.	<a href="#">Requirements Verification</a>	19
4.2.4.	<a href="#">Design Verification</a>	22
4.2.5.	<a href="#">Code Verification</a>	31
4.2.6.	<a href="#">Integration Verification</a>	31
4.2.7.	<a href="#">Documentation Verification</a>	32
4.3.	<a href="#">Validation of Neural Networks</a>	33
4.3.1.	<a href="#">Items Subject to Validation</a>	33
4.3.2.	<a href="#">Validation Environment</a>	33
4.3.3.	<a href="#">Testing</a>	34
4.4.	<a href="#">V &amp; V Metrics</a>	36
4.5.	<a href="#">Independent Verification and Validation (IV&amp;V)</a>	37
5.	<a href="#">APPENDIX A: ACRONYMS</a>	38
6.	<a href="#">APPENDIX B: GLOSSARY</a>	39
7.	<a href="#">APPENDIX C: FOR MORE INFORMATION</a>	42
8.	<a href="#">APPENDIX D: SELF ORGANIZING MAPS</a>	43
9.	<a href="#">APPENDIX E: SAMPLE REQUIREMENTS FOR THE PRE-TRAINED NEURAL NETWORK:</a>	46
10.	<a href="#">APPENDIX F: HESSIAN MATRIX</a>	50
11.	<a href="#">APPENDIX F: STABILITY COEFFICIENTS</a>	51
12.	<a href="#">APPENDIX G: INTELLIGENT FLIGHT CONTROL SYSTEM (IFCS)</a>	52
13.	<a href="#">APPENDIX H: V&amp;V ISSUES for NEURAL NETWORKS</a>	55
13.1.	<a href="#">Introduction</a>	55
13.2.	<a href="#">Notation</a>	56
13.3.	<a href="#">Data Ranges</a>	59
13.4.	<a href="#">Roundoff Errors</a>	59
13.4.1.	<a href="#">Accuracy of Operators</a>	60
13.5.	<a href="#">Scaling</a>	61
13.5.1.	<a href="#">Badly Scaled Problems</a>	61
13.5.2.	<a href="#">Influence of Scaling</a>	61
13.5.3.	<a href="#">How to Scale</a>	64
13.6.	<a href="#">Sensitivity Analysis</a>	65
13.7.	<a href="#">Condition Numbers</a>	67
13.8.	<a href="#">Analysis of the Training Algorithm</a>	71
13.8.1.	<a href="#">Progress of the Training</a>	72
13.8.2.	<a href="#">Stopping (the Training)</a>	74
14.	<a href="#">APPENDIX I: BASICS</a>	78
15.	<a href="#">APPENDIX J: QUADRATIC FUNCTIONS AND QUADRATIC FORMS</a>	80
16.	<a href="#">APPENDIX K: EIGENVECTORS AND EIGENVALUES</a>	82

<a href="#">17.</a>	<a href="#">REFERENCES</a> .....	84
---------------------	----------------------------------	----



# 1.NOMENCLATURE

The following nomenclature is used throughout this document:

- Pre Trained Neural Net (PTNN) refers to a supervised learning model that “learns” the relationship between inputs and outputs through training by a “teacher”. Once trained, the PTNN does not adapt or change during operation.
- Online Learning Neural Networks (OLNN) refers to neural nets that learn by adapting to regularities in data according to rules implicit in the design, but without a teacher; therefore, OLNN adapt or change during operation.
- Supervised/Unsupervised refers to the training method for the neural network where supervised neural networks require a teacher and unsupervised neural networks learn independently. Neural networks can be either supervised or unsupervised.
- Static/Dynamic refers to learning method of the neural network where static means the NN does not learn during operation and dynamic means the NN learns in real time during operation.

## 2.EXECUTIVE SUMMARY

The Dryden Flight Research Center V&V working group and NASA Ames Research Center Automated Software Engineering (ASE) group collaborated to prepare this report. The purpose is to describe V&V processes and methods for certification of neural networks for aerospace applications, particularly adaptive flight control systems like Intelligent Flight Control Systems (IFCS) that use neural networks.

This report is divided into the following two sections:

- Overview of Adaptive Systems
- V&V Processes/Methods

### 2.1.1. OVERVIEW OF ADAPTIVE SYSTEMS

Adaptive systems refer to systems that can assess a situation and make changes accordingly. The Intelligent Flight Control System (FCS) is an excellent example of an adaptive system because neural net software in the flight control system learns about the changes in the aerodynamics from sensors on or connected to aircraft surfaces (flaps, elevators, rudders, ailerons, et al) and provides vital updates to FCS to compensate in the case of a failure (stuck rudder, broken elevator, missing surface etc.)

The IFCS will use the neural networks listed below and explained in Section 3:

- Pre-trained Neural Net (PTNN)
- Online Learning Neural Network (OLNN)

This document concentrates on the V&V of PTNN and contains information known at time of publication about one type of OLNN called Dynamic Cell Structure (DCS). It does not provide a comprehensive discussion of V&V for OLNN. Research is underway at NASA to discover advanced V&V techniques to address the special and complex issues surrounding verification of OLNN.

### 2.1.2. V&V PROCESSES/METHODS

Software V&V is defined as the process of ensuring that software being developed or changed will satisfy functional and other requirements (verification) and each step in the process of building the software yields the right products (validation). In other words:

- Verification – Build the Product Right
- Validation – Build the Right Product

This section is divided into four parts:

- Verification
- Validation
- Metrics
- Independent Verification and Validation (IV&V)

#### Verification

Verification guidelines for Adaptive Systems are based on March 1998 IEEE/IEA 12207.0, paragraph 6.4. Types of verification to be conducted include:

- Contract verification – make sure that vendors of adaptive systems and independent contractors with expertise in adaptive systems follow standard contract guidelines

- Process verification - ensure that the project team includes neural net experts and V&V engineers with experience in testing safety critical systems including neural nets.
- Requirements verification
  - Verify the description of the learning algorithm
  - Verify that additional criteria for specifying the performance of the neural network exists
  - Verify accuracy of stopping criteria
  - Verify weights are properly adjusted for PTNN
  - Verify the topology of neural network
  - Verify the accuracy of the training set requirements
  - Verify the description of the desired output data, a range of those parameters, and the level of errors that is acceptable for adequate system performance
  - Verify correctness of each error range for the stability coefficients
  - Verify the appropriateness and use of stability proofs for the adapting algorithm
  - For subsequent implementations of a PTNN, verify that once trained, the weights are not altered as they are loaded into the system
- Design verification
  - First, ensure that the NN is trained on more and more complex training sets until it achieves a minimum error with a minimum number of weights. Then, make sure the weights are fixed and remain static for subsequent evaluation and implementation using either test or production data. Then, check that test data is used to verify the accuracy of the NN. Finally, compare that the output of the neural networks to the desired values to determine the minimum and maximum, as well as the average error.
  - Verify acceptable output ranges
  - Perform sensitivity analyses
  - Verify appropriate use of scaling
  - Examine the conditioning of the problem
  - Check stopping criteria
  - Test training time
  - Check the progress of training
  - Verify the training set
  - Verify that, when designing a NN, the learning characteristics are well understood
  - Verify that the learning algorithm was properly designed
  - Verify that the appropriate NN topology was designed

Additional design information can be found a book titled, *Neural and Adaptive Systems from Fundamentals through Simulations*.<sup>1</sup>

- Code verification - Code verification pertains to the NN software rather than the training of the NN. Traditional white box testing may be used.
- Integration verification
  - Verify that PID (parameter identification) data is in the appropriate form to match the PTNN data
  - Verify that the PTNN recalls proper aircraft model data and feeds it to DCS

- Verify that DCS receives PTNN and PID data in the proper form

**Note:** See Appendix E for more information about PID and other software that is integrated with the NN.

- Documentation verification - For the most part, descriptions of adaptive systems should be included in standard project documentation. Any special documentation about adaptive systems should be verified for technical accuracy by a peer review, then, catalogued and safeguarded following standard documentation configuration management procedures.

## Validation

Validation guidelines for Adaptive Systems are based on March 1998 IEEE/IEA 12207.0, paragraph 6.5. The validation process is documented in the validation plan and consists of the following:

- Items subject to validation – both PTNN and OLNN neural networks will be subject to validation to ensure they perform within acceptable ranges.
- Validation environment – Low, medium and high-fidelity testbeds will be necessary to properly test adaptive systems. Different versions of PTNN and DCS software are available for testing purposes.
- Testing to validate that the software satisfies its intended use.
  - Perform Unit Testing
  - Test the frequency response and phase and gain margins to ensure proper mil specs are met for stability
  - Perform failure modes and effects (FMEA) testing
  - Perform sensitivity analysis to ensure proper stability and flying qualities
  - Test timing to ensure that, under the worst case scenario, the system has adequate performance or degrades gracefully when saturated
  - Test system utilization (memory available, data through put and other system specific resources)
  - Check time to adapt for DCS and other OLNN. *Note: PTNN do not adapt*
  - Conduct a piloted evaluation

Additional validation activities for IFCS PTNN include:

- Checking interpolation of wind tunnel data for PTNN to check for gaps during training
- For PTNN, check points between knot data to make sure the curve is consistent (no unexpected jumps).
- List stability coefficients output from PTNN and compare against valid ranges
- Check that PTNN data sent to ground via telemetry is accurate to validate that the PTNN is working in the flight environment.
- Check that PID data is in the appropriate form (FIFO or average) to match PTNN data
- Compare training values with actual output values

## Metrics

Neural Network Metrics include:

- Learning time
- Recall response time

- Accuracy
- Repeatability
- Robustness in the face of failures

**Independent Verification and Validation (IV&V)**

An overview of Independent Verification and Validation (IV&V) requirements is included due to the possible impact on budgets for V&V of adaptive systems

## 3. OVERVIEW OF ADAPTIVE SYSTEMS

Adaptive systems refer to systems that learn about their environment and adjust accordingly. They can assess a situation, like a stuck rudder on an aircraft, and compensate for it. The Intelligent Flight Control System (IFCS) is an excellent example of this type of adaptive system because neural network software in the flight control system automatically assesses the aircraft state (rates, acceleration, air data and surface positions, et al) and provides flight information (stability coefficients) to compensate in the case of a failure (stuck rudder, broken elevator, missing surface etc.). Appendix G contains more information about IFCS.

Several types of neural networks (also called neural nets) were used in the IFCS. This section contains an overview of neural networks (NN) and a description of specific types of NN used in IFCS.

### 3.1. Overview of Neural Networks

A Neural Network (NN) is a collection of mathematical models that process information in a way that loosely mimics the human brain. They are used in modems, image processing/recognition systems, speech recognition systems, and adaptive aircraft flight control systems.

Neural networks differ from conventional computers in that they do not execute a set of predefined instructions. A NN is composed of a large number of highly interconnected processing elements called neurons that work in parallel to solve specific problems.

#### Typical Neuron

The human brain is composed of approximately ten thousand million highly connected units called neurons. These neurons are made up of four principal components consisting of the dendrites, soma, axon and synapses.

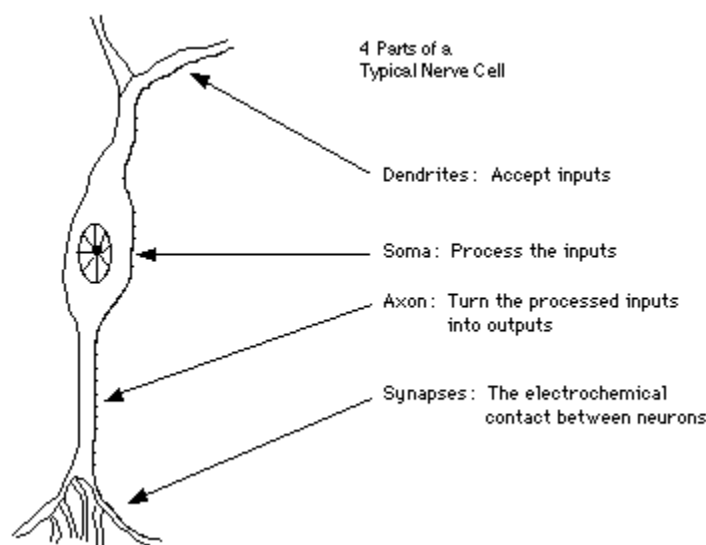


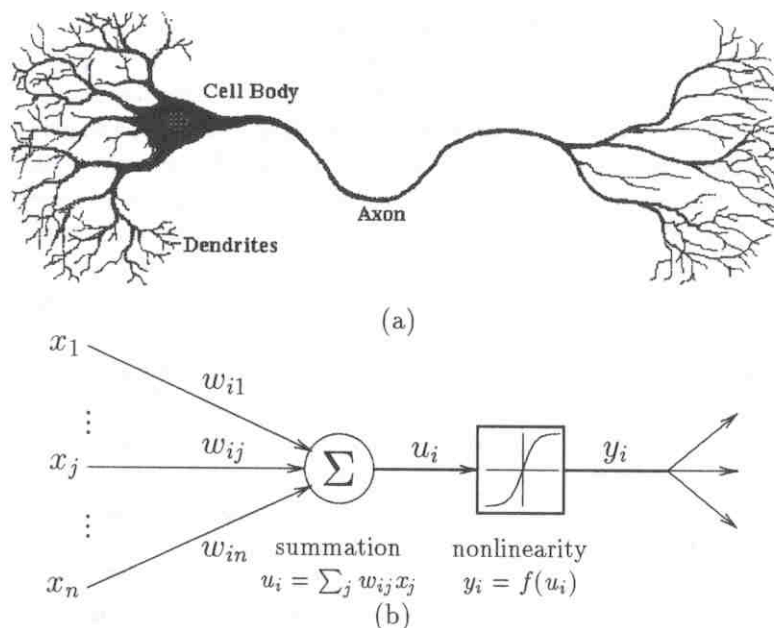
Figure 1: Typical Biological Neuron

The dendrites are the input conduits which receive input stimuli through the synapses of other neurons in the form of voltage spikes. They deliver electro-chemical excitation via hair-like nerve fibers to cell body, or soma, for processing. The soma processes the incoming signals in the form of sum and threshold functions. The processed values are output and distributed to other neurons through the axon and the synapses.

The axons connect to the dendrites of other neurons via synapses. When a voltage spike reaches a synapse, chemicals called neurotransmitters are released. These travel across the synapse gap and activate gates on the dendrites which produce a voltage pulse. Each dendrite may be connected to many synapses, some of which are excitatory and some inhibitory. Each neuron is connected to approximately ten thousand others that are grouped together into functional assemblies to perform specific tasks.

Learning occurs in the brain when the effective coupling between one cell and another is modified to reinforce good connections or remove bad ones. The axon links from the receptive neurons are affected by their distance from the activated neurons, causing cells which are physically close to the propagating cell, to have the strongest links. After a certain distance those links become inhibitory causing clustering of neurons into groups which respond to similar stimuli.

The following diagram compares a biological neuron to an artificial neuron like the feedforward perceptron used in the PTNN.



**Figure 2: Biological Neuron Compared to Artificial Neuron<sup>2</sup>**

As you see the biological (a) and artificial neurons (b) are very similar. Both receive input from a dendrite, process that input in the soma and axon and issue output via synapses. The input from the artificial dendrite is in the form  $x_1 \dots x_n$ . The artificial soma processes the input, by multiplying it by weights ( $w_{i1} \dots w_{in}$ ), then summing the products. The result of this summation is further processed via the artificial axon through a nonlinear function, in this case a tanh function (nonlinear functions are explained in Section 3.2). The output from the artificial synapse is  $y_i$ .

Learning is accomplished in various ways for different types of neural networks. The IFCS contains two types of neural nets:

- Pre-trained Neural Net (PTNN)
- Online Learning Neural Net (OLNN)

In the PTNN, learning occurs by discovering nodes and adjusting synaptic weights; therefore it can be configured for a specific application by learning from a set of training data.<sup>3</sup>

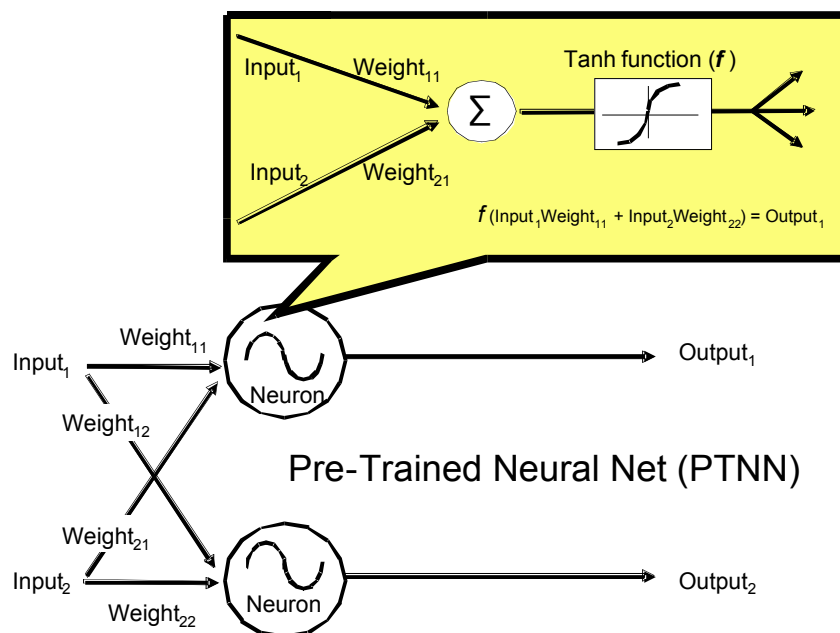
There are various types of OLNN including the Dynamic Cell Structure (DCS) neural net used in early generations of IFCS. Learning occurs in the DCS in much the same way as the human brain. Behavior or “edges” are reinforced or eliminated depending upon use and nodes are identified that best match the input data.

The following sections describe the PTNN and the DCS OLNN.

### 3.2. Pre-Trained Neural Networks (PTNN)

The Pre Trained Neural Networks (PTNN) used in the Intelligent Flight Control System (IFCS) is a supervised learning model meaning it “learns” the relationship between inputs and outputs through training by a “teacher”.

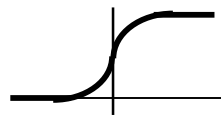
The process is referred to as “supervised” because an external “teacher” must specify the correct output for each and every input pattern. In the case of IFCS, the teacher is a model of the physical aircraft surfaces (flaps, rudders...). The PTNN learns through a process of determining the number of nodes (neurons) and adaptation of the weights (part of a mathematical computation shown in the following diagram).<sup>4</sup> When learning is complete, the weights are fixed so the PTNN can remember what it learned.



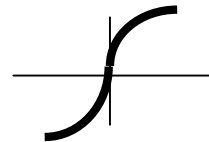
**Figure 3: Overview of PTNN**

Figure 1 shows how inputs to the neuron are weighted. It also shows the summation of the weighted inputs and one type of activation function called a hyperbolic tangent (tanh) used to compute the output. As shown below, the tanh function is a centered version of the sigmoid function. A sigmoid function is a continuous, bounded, monotonic function of its input  $x$ . It saturates at 0 for large negative inputs and at 1 for large positive inputs. Near zero, it is approximately linear.



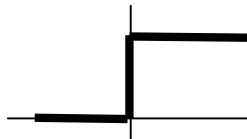


Sigmoid Function

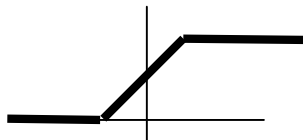


Tanh Function

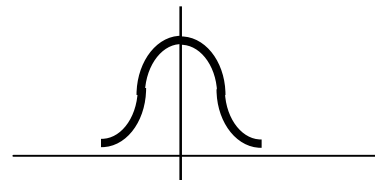
Other activation functions include the step function, ramp function and Gaussian function shown below<sup>5</sup>



Step Function



Ramp Function



Gaussian function

Since the weights are fixed, the pre-trained neural network receives input data and calculates the output within a given error, as defined by the weight values. This makes the PTNN easier to verify and validate than an unsupervised NN because the expected results (output) can be computed within an acceptable error margin.

**Note:** *Unsupervised means the NN learns by adapting to regularities in data according to rules implicit in its design, but without a teacher. Weights are not fixed in an unsupervised NN; therefore, they are more difficult to verify.*

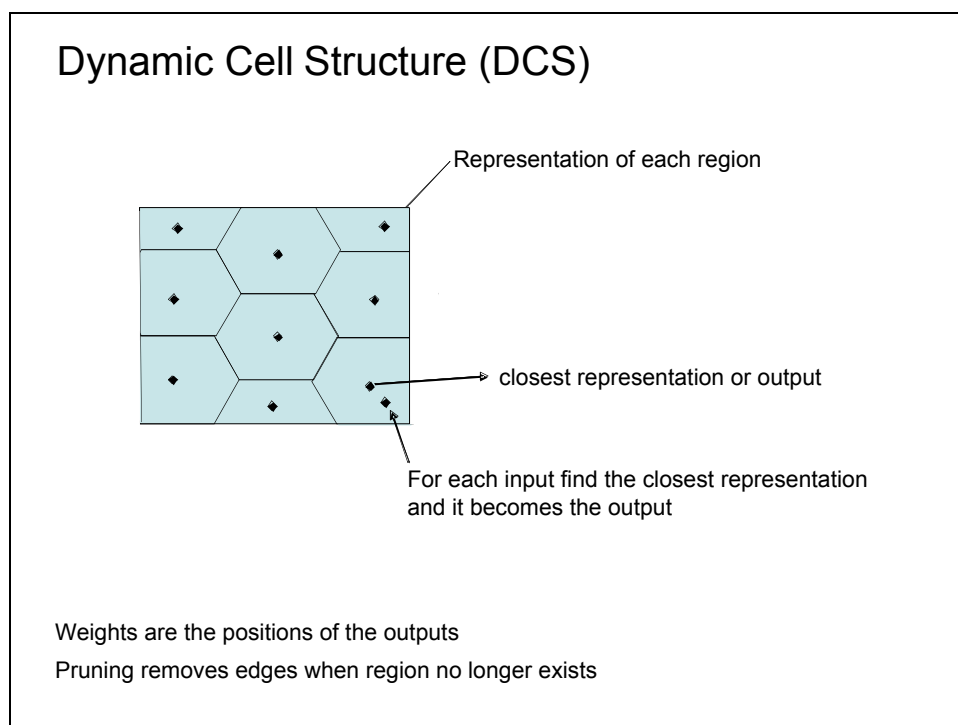
The IFCS PTNN has the following features:

- Multi-layer feedforward perceptrons (The term perceptron refers to a feedforward network of nodes with response like those shown in Figure 2 above. Feedforward means there are no connection loops that would allow output to feed back to their inputs and change the output at a later time. In other words, the network implements a static mapping that depends only on present inputs and is independent of previous system states.)
- Tanh neural activation function.

### 3.3. Online Learning Neural Networks (OLNN)

Online Learning Neural Networks adapt or change during operation. This process guide contains information known at publication regarding one type of OLNN, the Dynamic Dell Structure (DCS). It does not provide a comprehensive discussion of V&V for DCS or OLNN. Research is underway at NASA to discover advanced V&V techniques to address the special and complex issues surrounding verification of unsupervised, dynamic neural networks.

DCS is a Self Organizing Map (SOM). An overview of the DCS SOM is shown below and Appendix D contains an in depth technical discussion.<sup>6</sup>



**Figure 4: Overview of DCS** (input values and weights determine the position (value) of the outputs)

Nodes or neurons in DCS NN are points that represent regions of the surface being evaluated by the net. When the network receives input, it finds the node that is closest to the input and outputs that node. Therefore, the weights of a DCS NN used to map the output.

As it learns, regions in DCS change requiring modification to the edges of the regions called pruning. Pruning mechanisms are of particular interest for V&V because it is critical to evaluate points near the edges to ensure the proper region representative is selected for each point.

The IFCS DCS SOM has the following features:

- Uses block learning - receives several data samples at the same time and does a number of learning iterations on that data. An external module called SDB builds a block of data from the incoming stream
- Was designed with no global data and does not use any dynamic allocation calls

- Uses Hebbian learning (rather than competitive learning) - unsupervised neural networks learn based on the experience collected through the previous training patterns. Hebbian learning is one type of unsupervised learning. A simple version of Hebbian learning is that when unit  $i$  and unit  $j$  are simultaneously excited, the strength of the connection between them increases in proportion to the product of their activations.<sup>7</sup> This technique makes it possible to decay edges so that as the NN learns edges are either reinforced or eliminated.

Another type of unsupervised learning is called competitive learning. If a new pattern is determined to belong to a previously recognized cluster, then the inclusion of the new pattern into that cluster will affect the representation of the cluster. This will in turn change the weights characterizing the classification network. If the new pattern is determined to belong to none of the previously recognized clusters, then, the NN will be adjusted to accommodate the new class.

- Uses a Kohonen self organizing map to select the best matching node. The basic idea of a SOM is to incorporate into the learning process some degree of sensitivity with respect to the neighborhood or history. This provides a way to avoid totally unlearned neurons and helps enhance certain topological property which should be preserved in the feature mapping.

The question is how to train a network so the ordered relationship can be preserved. Kohonen allows the output nodes to interact laterally leading to the self-organizing feature.

First, a winning neuron is selected as the one with the shortest Euclidean distance between its weight vector and the input vector where  $w_i$  denotes the weight vector corresponding to the  $i$ th output neuron.

$$\|x - w_i\|$$

Second, let  $i^*$  denote the index of the winner and let  $I^*$  denote a set of indices corresponding to a defined neighborhood of winner  $i$ . Then the weights associated with the winner and its neighboring neurons are updated by

$$w_j = w_j + n(w - w_j)$$

For all the indices  $j$  is a member of  $I$  and  $n$  is a small positive learning rate. The amount of updating may be weighted according to a pre-assigned "neighborhood function". The convergence of the map depends upon the proper choice of  $n$ . One plausible choice is that  $n = 1/t$ . The size of the neighborhood should decrease gradually. The weight update should be immediately succeeded by the normalization of  $w_i$ .<sup>8</sup>

- Validation activities for DCS include:
  - Validate one version of DCS against other versions and check for same or better results
  - View graphs showing DCS output to visually inspect the NN output
  - Use traditional software techniques to perform white box testing for each module of DCS
  - Look at the entire boundary rather than just the corners and midpoints

## 4. V&V PROCESSES/METHODS

Software verification and validation (V&V) is defined as the process of ensuring that software being developed or changed will satisfy functional and other requirements (verification) and each step in the process of building the software yields the right products (validation). In other words:

- Verification – Build the Product Right
- Validation – Build the Right Product

Current V&V practices for adaptive systems follow NASA V&V standards for traditional software with special consideration at each Life Cycle stage for the neural net software. Therefore, this section is divided into the following subsections:

- Overview of applicable NASA standards for V&V of airborne software
- Enhancements to *verification* guidelines (based on March 1998 IEEE/IEA 12207.0, paragraphs 6.4 and 6.5) for V&V of Adaptive Systems
- Augmentation of *validation* guidelines based (based on March 1998 IEEE/IEA 12207.0, paragraph 6.5) for Adaptive Systems
- V&V Metrics
- Overview of Independent Verification and Validation (IV&V) requirements and the possible impact on budgets for V&V of adaptive systems

### 4.1. Overview of NASA V&V Standards

Applicable NASA V&V Standards include the two documents listed below:

- NASA Guidebook for Safety Critical Software, NASA-GB-1740.13-96
- Trial-Use Standard for Information Technology Software Life Cycle Processes - Software Development, J-STD-016-1995
- Dryden center policy, D C P-S-007
- IEEE Standard for Software Test Documentation, IEEE Std 829-1998 (Revision of IEEE Std 829-1983)
- NASA Procedures and Guidelines (NPG) 2820.DRAFT, NASA Software Guidelines and Requirements<sup>9</sup>
- NASA Procedures and Guidelines (NPG) 8730.DRAFT 2, Software Independent Verification and Validation (IV&V) Management<sup>9</sup>

NPG 2820.DRAFT references the following IEEE/EIA Standards<sup>10</sup> developed in accordance with ANSI:

- 12207.0 - *Standard for Information technology – Software Life Cycle Processes (March, 1998)*
- 12207.1 - *Standard for Information technology – Software Life Cycle Data (April, 1998)*
- 12207.2 - *Standard for Information technology – Software Implementation Considerations (April, 1998)*

The IEEE documents reference the ISO and IEC standards published as ISO/IEC 12207 in 1995.

In addition to the NASA standards, DO-178B, "Software Considerations in Airborne Systems and Equipment Certification" contains guidance for determining that software aspects of airborne systems and equipment comply with airworthiness certification requirements. Written in 1980 by the Radio Technical Commission for Aeronautics (now RTCA, an association of aeronautical organizations of the United States from both government and industry), it was revised in 1985 and again in 1992. During the 1992 revision, it was compared with international standards: ISO 9000-3 (1991), "Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software" and IEC 65A (Secretariat) 122

(Draft – 11-1991), “Software for Computers in the Application of Industrial Safety-Related Systems” and considered to generally satisfy the intent of those international standards.

#### NASA V&V Standards Acronyms:

ANSI - American National Standards Institute  
 EIA - Electronic Industries Association  
 IEC - International Electro-technical Commission  
 IEEE - Institute of Electrical and Electronics Engineers  
 ISO - International Organization for Standardization  
 NPG - NASA Procedures and Guidelines

### 4.1.1. Life Cycle

The following diagram shows the relationship between V&V and the Life Cycle Phases described in the standards.<sup>9</sup>

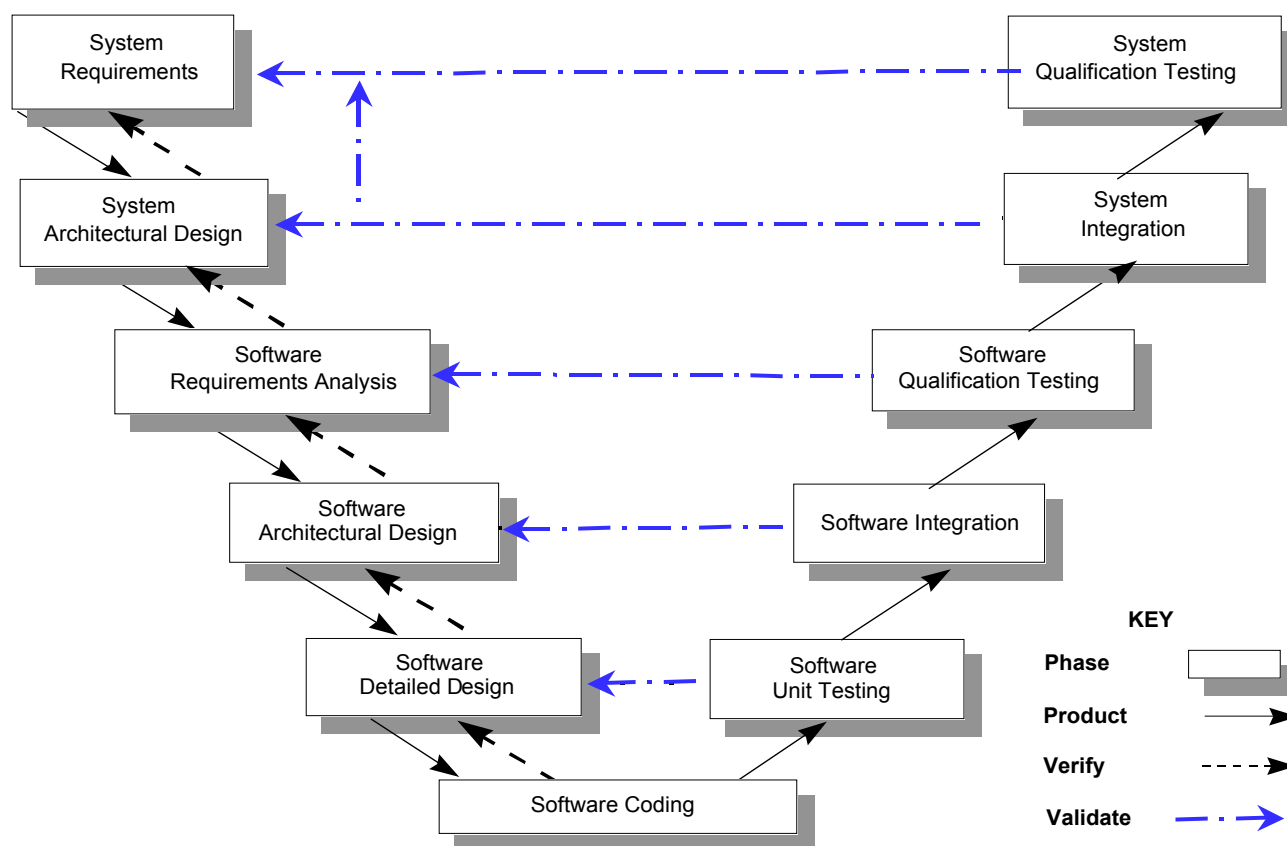


Figure 5: V&V for Life Cycle Phases

The following list contains an overview of additions to each Life Cycle phase for V&V of adaptive systems:

- Systems requirements must be enhanced to include NN specification

- System Architectural Design must contain NN architecture including integration with other systems
- Software Requirements Analysis must include NN software requirements including type of NN, learning algorithm, a description of the inputs and outputs, acceptable errors and training set(s) for pre-trained NN
- Software Architectural Design should contain NN software architecture design including type of NN (feedforward, Self Organizing Map, etc) and the learning algorithm (Least Means Squared (LMS), Levenberg-Marquardt, Newton's method etc)
- Software detailed design must include a description of precise code constructs required to implement the NN
- Software coding must contain NN code
- Unit testing must include both black and white box testing for modularized NN code
- Software integration should verify that the NN interfaces with other software including proper inputs and outputs for the NN
- Software Qualification Testing should ensure that the requirements are sufficiently detailed to adequately and accurately describe the NN
- System integration testing should verify that the architectural design is detailed enough so, when implemented, the NN can interface with system hardware and software in various fidelity testbeds
- System qualification testing should verify that the system requirements are sufficient enough to ensure that, when implemented, the NN will interface properly with the system in production

## 4.2. Verification of Adaptive Systems

Verification is the process in which a piece of embedded software, such as a neural network, is tested against its specification. Verification guidelines for Adaptive Systems are based on March 1998 IEEE/IEA 12207.0, paragraph 6.4. Types of verification to be conducted include:

- Contract
- Process
- Requirements
- Design
- Code
- Integration
- Documentation

For sake of completeness, each type of verification is discussed in this section, but only as it applies to V&V of adaptive systems.

### 4.2.1. Contract Verification

Contract verification includes the following steps to ensure the contracts are managed effectively:

- Verify that supplier has capability to satisfy requirements
- Ensure that requirements are consistent and cover user needs
- Provide adequate procedures for handling changes to requirements and escalating problems
- Provide procedures for interface and cooperation among the parties
- Ensure acceptance criteria and procedures are stipulated in accordance with requirements

Contracts for vendors of adaptive systems and independent contractors with expertise in adaptive systems must follow these guidelines.

#### 4.2.2. Process Verification

Process verification ensures that project planning is adequate, processes are compliant with the governing contract and processes are being executed. It makes sure that standards, procedures and environments for the project are adequate and that the project is staffed with trained personnel.

Each of these items pertain to adaptive systems, particularly ensuring that the project team includes neural net experts and V&V engineers with experience in testing safety critical systems including neural nets.

#### 4.2.3. Requirements Verification

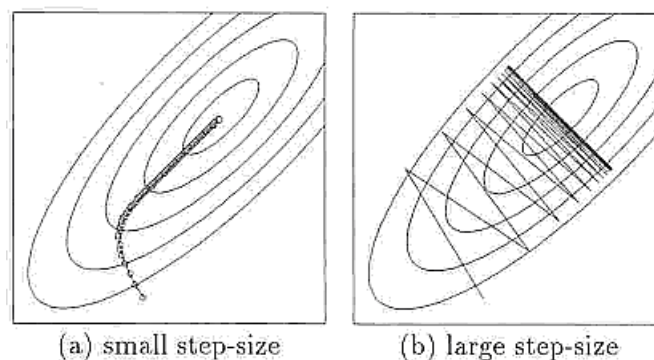
Requirements verification will make sure that:

- System requirements are feasible, consistent and testable
- Requirements have been appropriately allocated to hardware items, software items and manual operations according to design criteria
- Software requirements related to safety, security and criticality are correct as shown by suitably rigorous methods

Requirements for adaptive systems are different than traditional system requirements because neural nets have the ability to learn. However, it is possible to verify these requirements using the following techniques:

- Verify the description of the learning algorithm. The learning algorithm is used to adapt to neural network to minimize the error between the desired output and the actual output. A common approach is to calculate a mean squared error of the differences between the desired output and the actual outputs. **Note:** *A low mean squared error doesn't mean the NN has been trained adequately over the entire training envelope.* Other approaches include Newton methods and Levenberg-Marquardt.

To understand LM, one must first know about gradient descent and Newton's method. These two methods are used to find the global minimum and employ different methods for selecting a direction and step size. The following diagrams show challenges when selecting the step size and the problems that may occur when the step size is too small or too large. Using a small step size (a), the neural net may crawl to the global minimum but not meet response time requirements. Using a large step (b) size may result in approaching the minimum but churning back and forth.



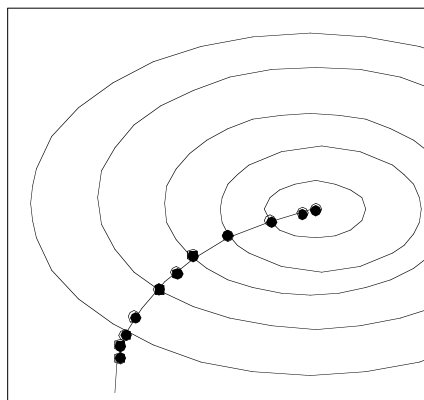
Gradient descent selects direction and step size like a “good skier going down a steep hill”. It picks the steepest point on the hill and descends. Unlike the skier, however, gradient descent converges very slowly, so faster methods have been devised including Newton and Levenberg-Marquardt.

Newton’s method is the theoretical standard by which other optimization methods are judged. Because it uses all the first and second order derivative information in its exact form, its local convergence properties are excellent. Unfortunately, it is often impractical because explicit calculation of the full Hessian matrix (explained in Appendix F) can be very expensive in large problems.

The Levenberg-Marquardt method (shown in Figure 6 below) is a compromise between Newton’s method which converges quickly near a minimum but may diverge elsewhere and gradient descent which converges everywhere slowly. The LM search direction is linear combination of the steepest descent direction and the Newton direction  $H^{-1}g$ :

$$w_{k+1} = w_k - (H + \lambda I)^{-1} g$$

Parameter  $\lambda$  controls the compromise. This can be viewed as forcing  $H + \lambda I$  to be positive by adding a scaled identity matrix. The minimum value of  $\lambda$  needed to achieve this depends on the eigenvalues of  $H$ . The algorithm starts with  $\lambda$  large and adjusts it dynamically so that every step decreases the error. Generally it is held near the smallest value that causes the error to decrease. In the early stages when  $\lambda$  is large, the system effectively does gradient descent. In later stages,  $\lambda$  approaches 0, effectively switching to Newton’s method for final convergence.<sup>11</sup>



**Figure 6: Levenberg-Marquardt**

- For flight critical applications, verify that additional criteria for specifying the performance of the neural network exists
- Verify accuracy of stopping criteria (when the NN stops learning). Stopping criteria may include:
  - A specified error or error range



- Specific number of learning iterations
- Verify weights are properly adjusted for PTNN. The network output is a function of its weights. Consider how weights should be adjusted to minimize error.
- Verify the topology of neural network. The topology can be either single or multi-layer perceptrons. Perceptrons refer to any feedforward network of nodes with responses in the following equation<sup>12</sup>

$$y = f\left(\sum_k w_k x_k\right)$$

**Note:** While, it is mathematically possible to determine the topology of the neural network to perform a given function, more often, the topology is set by watching the learning curve as the training and cross validation data are applied. This is an iterative process.

- Verify the accuracy of the training set requirements. Verify that the number and type of parameters and the range of the parameters are specified. In order to properly train the NN, the amount of training data must be sufficient to train the number of weights that exist in the network. The training data must also consider all parameters that may affect performance of the network. For example, in the case of landing; Mach, altitude, and angle of attack are some key parameters. However vehicle configuration parameters, such as flap setting and gear position, are also important.
- Verify the description of the desired output data, a range of those parameters, and the level of errors that is acceptable for adequate system performance. During training, the NN output should be compared to the desired value to determine the maximum, mean and minimum error.
- Verify correctness of each error range for the stability coefficients
- Verify the appropriateness and use of stability proofs for the adapting algorithm. IFCS plans to use the Lyapunov Stability Proof. A Lyapunov function has the following properties that can be verified:
  - The function must be continuous and have a continuous first partial derivative over the domain
  - The function is strictly positive except at the equilibrium point
  - The function is zero at the equilibrium point
  - The function approaches infinity at infinity
  - The function has a first difference that is strictly negative in the domain except at the equilibrium point<sup>13</sup>
- For subsequent implementations of a PTNN, verify that once trained, the weights are not altered as they are loaded into the system. This occurs when a legacy PTNN (like the PTNN used for IFCS F-15 experiment) is used. In this situation, it is important to verify that new implementation of the PTNN matches preceding implementations.

Appendix E contains a sample specification for a PTNN that was flown as Class B software (experimental vehicles), using a limited, up and away flight envelop. It provides some of the information required in the specification for a Class A (manned flight vehicles) NN.

#### 4.2.3.1. Requirements Traceability

A tracking system for adaptive systems must contain not only the capability to track traditional requirements, but also, enhanced traceability tools and techniques for neural nets.

**Note:** Most specifications have a verification and validation cross-reference matrix identifying tests required.

#### 4.2.4. Design Verification

Design verification makes certain that:

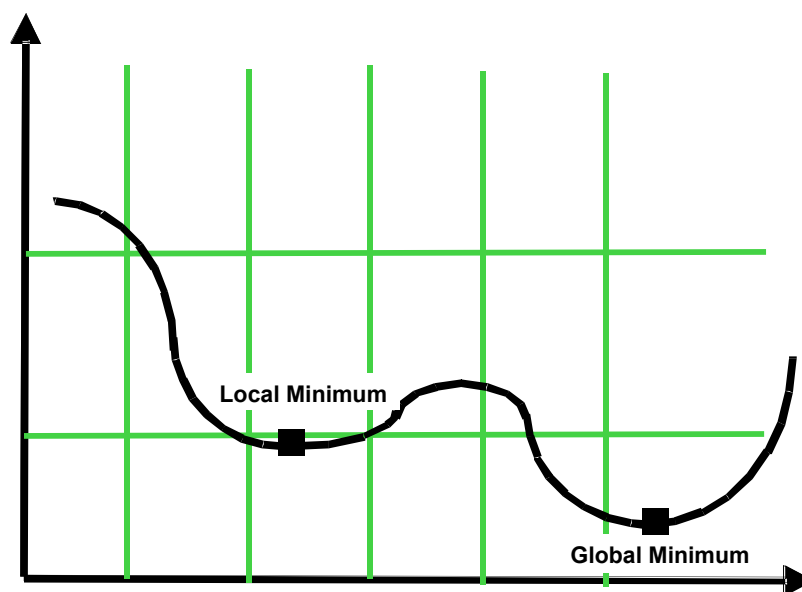
- Design is correct
- Design is consistent with and traceable to requirements
- Design implements proper sequence of events, inputs, outputs, interfaces, logic flow, allocation of timing and sizing budgets and error definition, isolation and recovery
- Selected design can be derived from requirements
- Design implements safety, security and other critical requirements as shown by suitable rigorous methods

The design of adaptive systems is different than the traditional waterfall methodology, but very similar to the iterative design techniques used in newer software development. For example, the iterative approach to software design requires building more and more complex prototypes of the system until it functions properly. The final prototype is then made production ready and implemented.

A typical NN design approach is very similar:

- First, the NN is trained on more and more complex training sets until it achieves a minimum error with a minimum number of weights. Then, the weights are fixed and remain static for subsequent evaluation and implementation using either test or production data.
- Then, test data is used to verify the accuracy of the NN. To ensure unbiased testing of the NN, a test data set that is independent of the training data is used. The error metrics used (i.e. mean square, maximum absolute...) in the NN testing should yield results comparable to those obtained during training. In addition to verifying the training set, the internal NN test data must be recorded and analyzed. The learning curve must be monitored to ensure that the neural networks are not stuck at a local minimum.

In order to understand a local minimum, a global minimum must be defined. A global minimum of a function can be defined as its lowest point (the input that gives the lowest possible output). A local minimum is a point that is lower than all surrounding points, but higher than the global minimum.



**Figure 7: Global versus Local Minimum (in two dimensions)**

- Finally, the output of the neural networks must be compared to the desired values to determine the minimum and maximum, as well as the average error. Depending on the specific type of neural network, additional internal parameters or metrics may need to be recorded to ensure proper response.

Verification of the NN design process is divided into the following categories:

- Mathematical Approach to Verification of PTNN
- Verification Processes/Methods for PTNN
- Definition of the Training Set
- Other Design Process Activities

#### **4.2.4.1. Mathematical Approach to Verification of PTNN**

This section provides an overview of mathematical approach. An in depth discussion is contained in Appendix H, *V&V Issues for Neural Networks* by Johann Schumann, NASA Ames Research Center.

The following mathematical analysis should be performed during the design of the PTNN:

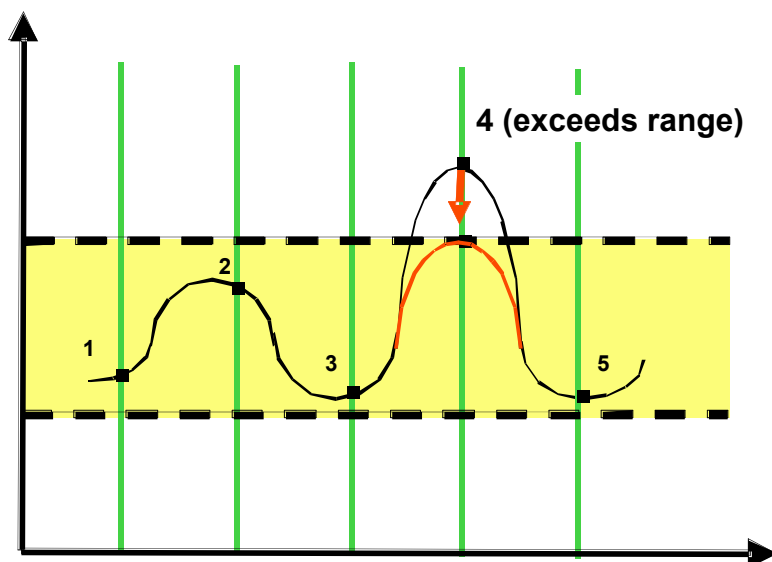
- Verify acceptable Output Ranges
- Sensitivity Analyses
- Scaling
- Conditioning of the Problem
- Stopping Criteria
- Training Time
- Progress of Training

#### **Acceptable Output Ranges**

Check the PTNN engine to ensure output falls within acceptable ranges (as defined in the requirements) for the given input. For the PTNN, it is important to verify the output by listing stability coefficients (output from PTNN) and compare them against valid ranges for each coefficient. Appendix F contains a list of sample stability coefficients.

It is also critical to compare training values for the PTNN with actual output values.

For output outside the acceptable range, a technique called “clipping” may be used to eliminate these outliers. As shown in the following diagram, clipping is a technique whereby if an output exceeds a pre-determined range (blue bar); the output equals the maximum of that range.<sup>14</sup>



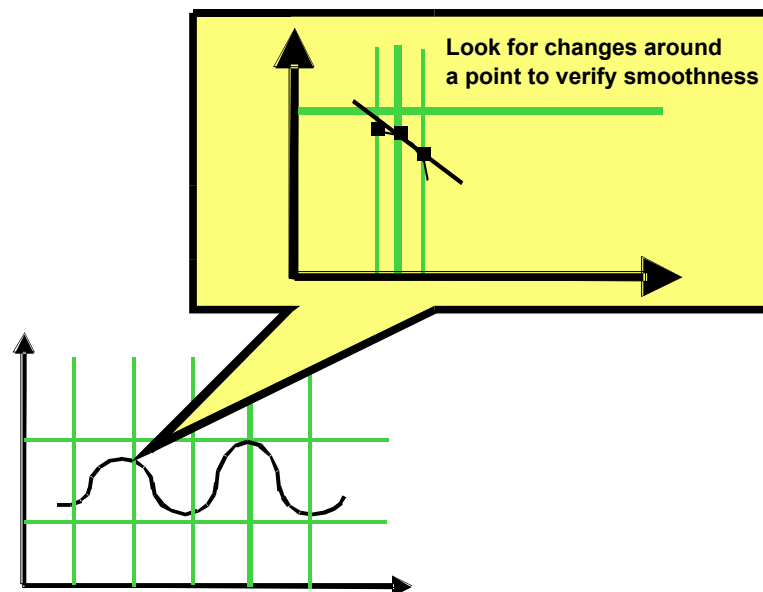
**Figure 8: Clipping**

The blue bar indicates a pre-determined range. As you can see the neural net output (black curve) exceeds this range around the fourth point. Therefore, clipping allows this point to be reset to the top of the desired range and the curve to be adjusted. This eliminates the outlier and increases system predictability. Verify if and how clipping was implemented and test clipped points to make sure they fall within the desired range and yield appropriate output. Also, test surrounding points using sensitivity analysis (explained below) to ensure that surrounding points do not exceed the maximum range.

For more information about data ranges see Appendix H.

### **Sensitivity Analysis**

Sensitivity analysis tests the smoothness of the curve at various intervals from a point in the NN. Derivatives may be calculated and graphed showing the changes in the curve. Sensitivity analysis is most valuable for critical data like wind tunnel data used to train the PTNN. By checking smoothness it is possible to find anomalies that would otherwise remain hidden.



**Figure 9: Sensitivity Analysis (Smoothness)**

Sensitivity Analysis is conducted by computing two derivatives. The 1<sup>st</sup> Derivative is the change in output over the change in input plus/minus the error:

$$\partial \text{ output} / (\partial \text{ input} \pm \_)$$

The second derivative is the change in the first derivative:

$$\partial / \partial$$

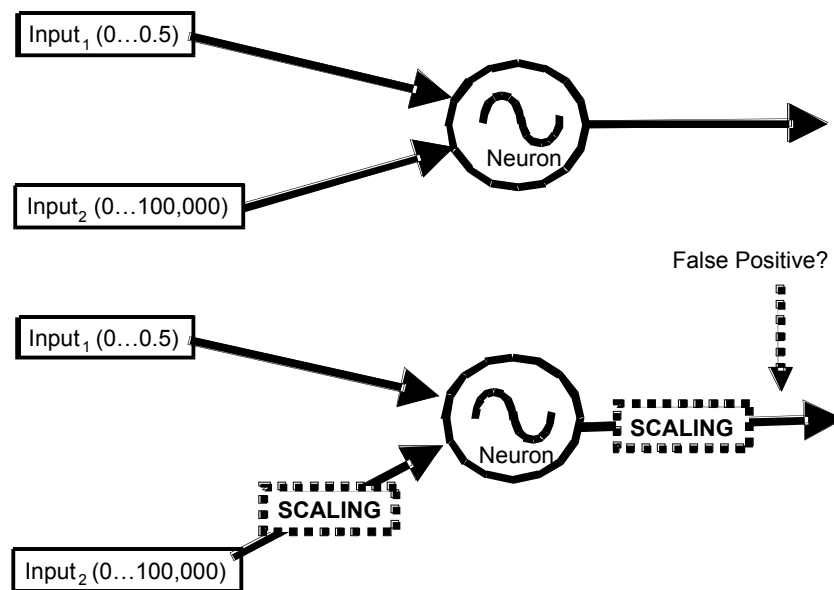
Sensitivity Analysis should be applied to the following:

- Checking interpolation of wind tunnel data for PTNN to check for gaps during training
- Testing check points between knot data in the PTNN to make sure the curve is consistent (no unexpected jumps)
- Look at the entire boundary (or a sufficient sample thereof) rather than just the corners and midpoints

For more information about sensitivity analysis see Appendix H.

### Scaling

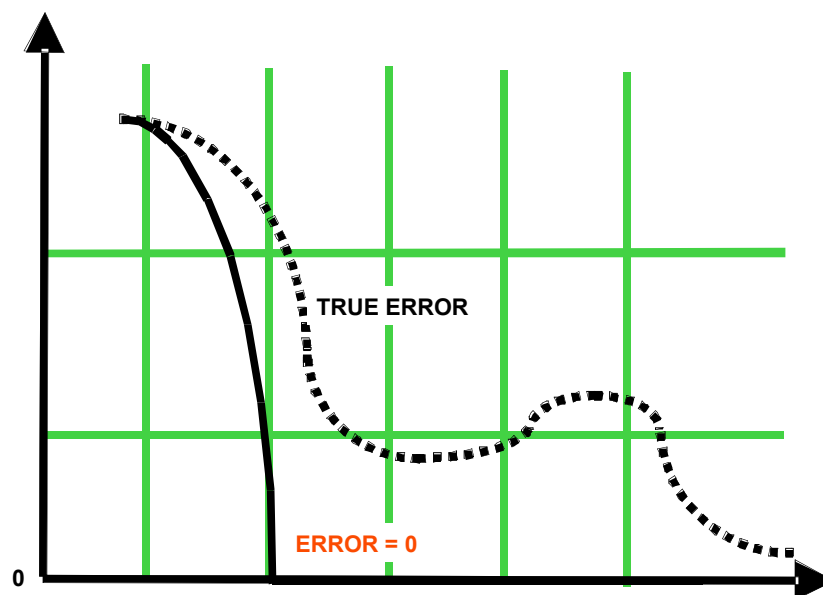
Sometimes NN inputs are different by orders of magnitude. When this occurs, it is difficult to determine the true error. The example below contains very small Input<sub>1</sub> data in the range of 0 to 0.5 and very large Input<sub>2</sub> data in the range of 0 to 100,000. The resulting error is so small that it cannot be computed; and therefore, becomes zero leading false belief that the NN was properly trained. Therefore, it is important to verify that the scaling techniques used do not result in a false positive. An example is shown in the following diagram:



For more information about scaling see Appendix H and *Neural and Adaptive Systems* by Principe, et al.<sup>15</sup>

### Conditioning of the Problem

Conditioning of the problem combines sensitivity analysis and scaling. It is required when the possibility exists that due to the nature of the input data (for example:  $\text{input}_1$  very small and  $\text{input}_2$  very large), the error rate could become zero when the true error is not. The following figure shows a graphical example.



### Stopping Criteria

Stopping criterion relates to when the neural net should stop learning. There are several types of stopping criteria:

- When the actual error is less than a specified error
- When learning iterations equal a specified number
- By checking the speed of convergence. Convergence results when learning stalls meaning very little learning occurs after each iteration.

For more information about scaling see Appendix H.

### Training Time

Training time is the time it takes a NN to be trained. It is important to optimize the training time by making trade-offs between training speed and accuracy. The following factors may cause training to be slow:

- Size and distribution of the training because each pass through the data takes twice as long when the data is twice as big.
- Irrelevant data (additional patterns that contain no new information) make training slower and provide no additional value to the learning process
- Overly restrictive convergence criteria. The acceptable error range must be chosen carefully so as to maximize learning and minimize training time.
- Paralysis due to sigmoid saturation. Sigmoid and related functions have nearly flat tails where the derivative is approximately zero for large inputs. Because  $\sigma'$  is proportional to the slope  $f_i$ , this leads to small derivative for weights feeding into the node and so on backward through the network. If many nodes are saturated, then weight derivatives may become very small and learning will be slow. In digital simulations, deltas may become so small that they are quantized to zero and learning stops. *Note: double precision arithmetic is sometimes recommended for this reason.*
- Flat regions in the error surface where the gradient is small
- Ill-conditioning of the Hessian matrix (explained in Appendix F.). An ill-conditioned matrix indicates that gradient changes slowly along one direction and rapidly along another, similar to a narrow ridge along the top of a mountain. A small learning rate must be used to avoid instability along the quickly changing direction to prevent falling off the ridge; however, this small step size will result in sluggish progress along the slowly changing direction and long convergence times.
- Poor choice of parameters such as learning rate (modification to the back-propagation weight update) and momentum (a common modification of the basic weight update rule to stabilize the weight trajectory by making the weight change a combination of the gradient-decreasing term plus a fraction of the previous weight change. Momentum gives a certain amount of inertia since the weight vector will tend to continue moving in the same direction unless opposed by the gradient term. Momentum can help avoid a local minimum because the inertia can push over the local maximum and continue towards the global minimum.)<sup>16</sup>
- Use of simple gradient descent methods when more sophisticated methods are more efficient
- Global nature of sigmoid functions. A change in one weight may alter the network response over the entire input space. This changes the derivative fed back to every other weight and produces further weight changes whose effects reverberate throughout the network. It takes time for these interactions to settle.
- Poor network architectures. The minimal size network just adequate to represent the data may require a very specific set of weights that may be very hard to find. Larger networks may have more ways to fit the data and so may be easier to train with less chance of convergence to poor local minima.

#### 4.2.4.2. Verification Processes/Methods for PTNN

The following processes and methods should be performed during the design of the PTNN:

- Verify the training set. The size of the training set can be computed using the following generalized formula where the size of the training set and the number of weights, are related.

<p style="text-align: center;"><b><math>N &lt; W/e</math></b></p> <p><math>N</math> is the number of training patterns in the training data</p> <p><math>W</math> is the number of weights in the NN. <b>Note:</b> <i>There is one weight for each input that feeds each neuron. Networks with two hidden layers have more weights than a single hidden layer</i></p> <p><math>e</math> is the value of error allowed</p> <p>For a 1% error, <math>N</math> should be 100 times the number of weights</p>
---

The training data should cover the entire domain to be learned and it must be representative of the data to be used in the test or in production.

- Verify that, when designing a NN, the learning characteristics are well understood. Learning is directly affected by:
  - Initialization of the weights
  - The learning rate or rate of change in values of the weights
  - The learning algorithm that finds the minimum error. Many learning algorithms exist, from the basic Least Means Squared (LMS) to more computational intensive methods such as Levenberg-Marquardt and Newton's method. The best method to use is based on the complexity of the problem, the time available for the network to learn an optimum solution, and other factors. Refer to (Principe 2000) for detailed explanations.
- Verify that the learning algorithm was properly designed. The PTNN uses Levenberg-Marquardt
- Verify that the appropriate NN topology was designed. NN topology affects the number of weights, and therefore, is related to the expected error. When considering NN topology, think about how a well-trained network will perform based on its test data set. The performance of a trained network to test data is referred to as generalization. It is intuitive that if a NN is to learn a function it must have a sufficient number of weights to do so. This may lead one to believe that a larger training set is better, but this is not true.

A NN with too many weights, may not generalize well to the test data or in production. There is no simple answer to designing the correct topology. There are two basic approaches:

- Begin with a small network, evaluate it, then grow as required
- Start with a large network and shrink until the desired result is achieved

Some design tools use automatic or genetic methods to help find the optimum topology.

#### Verification Used in Flight Test of Class B PTNN

The following design recommendations are based on experience with the flight test of a Class B experimental aircraft using a PTNN to approximate 26 baseline aircraft stability coefficients and control derivatives. The figure below shows the envelope in which the neural network operates (smaller envelope) verses the larger "ACTIVE" aircraft envelope in which the neural network was trained and tested.



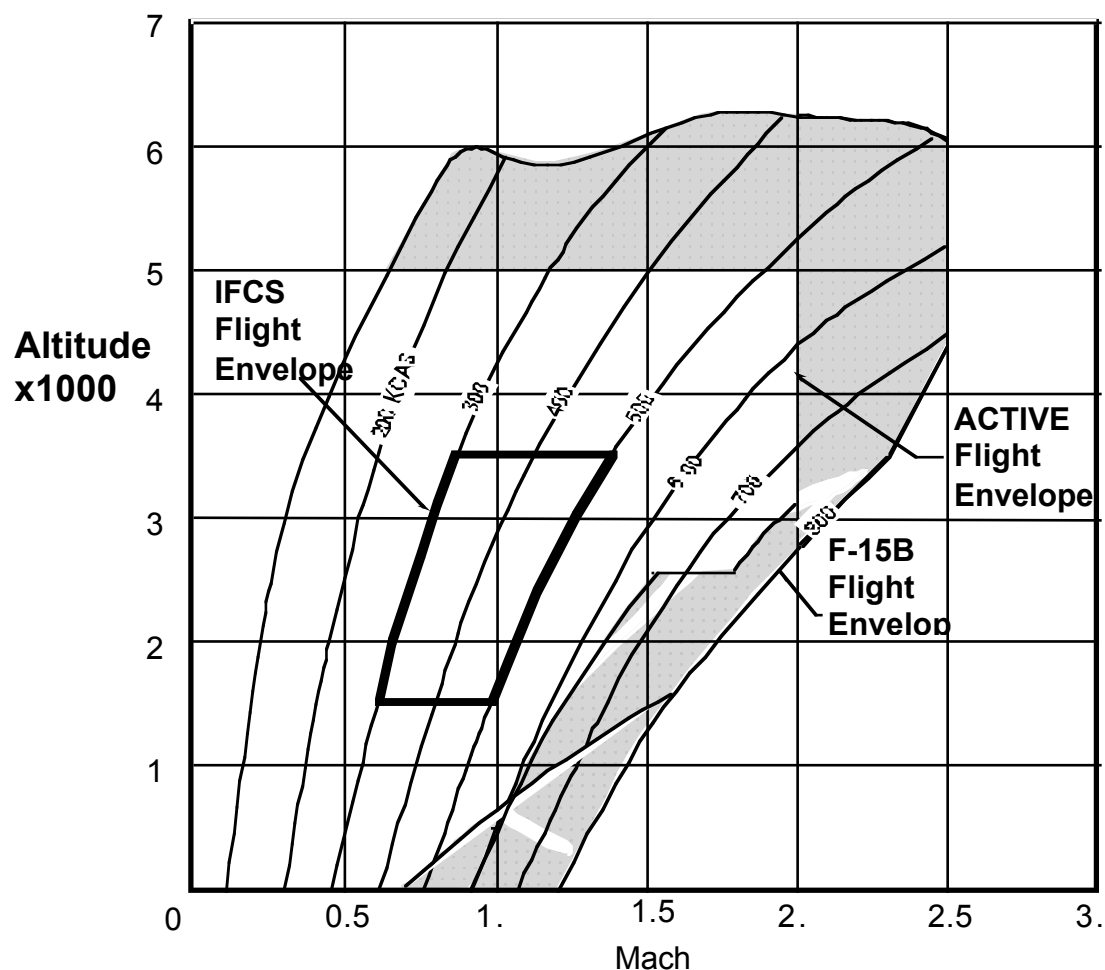


Figure 10: F-15B Flight Envelope

The following verification techniques were used to ensure that the design yielded a safe, secure system in compliance with requirements:

- First, the definition of the training data was considered. Training data for aircraft stability coefficients included:
  - Mach
  - Altitude
  - Alpha (Angle of attack)
  - Angle of side slip
  - Control surface deflections

A good understanding of aeronautics was required to identify which independent variables are used for each stability coefficient. The number of independent variables for each coefficient ranged from three to five. The variables are described in Appendix E.

- Next, the design was checked to ensure testability by checking that the software was modular and that each module could be tested individually, as well as, part of the whole system. Testability also requires the instrumentation of internal parameters so that they can be recorded and

analyzed during test. Testing of all possible paths through the software, usually based on logical settings, is also required. The primary purpose of all this testing is to verify the software meets its specification and this system operates safely.

#### 4.2.4.3. Definition of Training Set

Defining the training set for a neural network is the one of the most important steps. The training data consists of the variables used on the input as well as the desired output of the system. In the case of the PTNN, the output is a known mathematical function of the required input variables. The following table shows a partial data set used to train the neural network. It is a function of three input variables and one output variable which yields good results using a single layer neural network.

The variables are:

- MACH is the velocity in terms of MACH number (input)
- ALP is Alpha or angle of attack (input)
- BETA is the angle of side slip (input)
- CZ2 is a stability coefficient (output)

**Sample Training Set**

MACH	ALP	BETA	CZ2
0.2	-4	0	0
0.2	-4	2	0.000371
0.2	-4	5	0.00093
0.2	-4	7.5	-0.001758
0.2	-4	10	-0.004445
0.2	-4	12.5	-0.005766
0.2	-4	15	-0.007087
0.2	-4	20	-0.005062
0.2	-2	0	0
0.2	-2	2	0.000782
0.2	-2	5	0.001956
0.2	-2	7.5	0.002149
0.2	-2	10	0.002343
0.2	-2	12.5	0.004838
0.2	-2	15	0.007335

Defining the test data for the neural network is the second step. In some applications, the data that describes the input and output relationship, is broken down into the training set and the test set. In our example with the PTNN, we have a detailed simulation of the vehicle which allows us to have an extensive training and test sets. The table below gives a brief description of the test set used for the PTNN. Notice the difference in the values verses the training set.

**Sample Test Data**

MACH	ALP	BETA	CZ2
0.2	-4	0	0
0.2	-4	1	0.000185
0.2	-4	3.5	0.00065
0.2	-4	6.25	-0.000415
0.2	-4	8.75	-0.003103
0.2	-4	11.25	-0.005108

0.2	-4	13.75	-0.00643
0.2	-4	17.5	-0.00608
0.2	-4	20	-0.005069
0.2	-3	0	0
0.2	-3	1	0.000287
0.2	-3	3.5	0.001006
0.2	-3	6.25	0.000812
0.2	-3	8.75	-0.00044
0.2	-3	11.25	-0.000775
0.2	-3	13.75	-0.000193
0.2	-3	17.5	0.003245

#### 4.2.4.4. Other Design Process Activities

In order to test the PTNN, it may be necessary to design and develop NN verification tools in parallel, but independent from the primary developers. Such tools may include a NN simulator and enhanced requirements tracking tools specifically for NN verification.

#### 4.2.5. Code Verification

Code verification ensures that:

- Code is traceable to design and requirements, testable, correct, and compliant with requirements and coding standards
- Code implements proper event sequence, consistent interfaces, correct data and control flow, completeness, appropriate allocation timing and sizing budgets and error definition isolation and recovery
- Code can be derived from design or requirements
- Code implements safety, security and other critical requirements correctly as shown by suitably rigorous methods

Code verification pertains to the NN software rather than the training of the NN. Neural nets have traditionally been viewed as a black box; however, NASA researchers modularized NN code so that it can be considered a white box and tested accordingly. Specific tests are included in Section 4.3.3.

During code verification, enhancements may be required to make requirements tracking tools robust enough to clearly tie requirements to neural net code. For example, visual inspection of graphs provides an excellent way to verify NN code. However, typical tracking tools may not have a mechanism for numbering or storing graphics or charts as proof of verification efforts. Additionally, it may be necessary to store a special graphics viewer with a graphics file to ensure they can be viewed in the future.

#### 4.2.6. Integration Verification

Integration verification makes sure that:

- Software components and units of each software item have been completely and correctly integrated with hardware items
- Software items and manual operations of the system have been completely and correctly integrated
- Integration tasks have been performed in accordance with an integration plan

NN are generally an integral part of another system. For example, the NN on the IFCS is an integral part of the flight control system. Therefore, it is important to verify that all inputs and outputs between systems

are properly scaled and that the NN properly interfaces with the control system. The following are key items to verify for IFCS/NN integration:

- The PTNN gets sensor information from the PID (Parameter Identification). Verify that PID data is in the appropriate form to match the PTNN data; otherwise, incoming sensor data will be compared to a dissimilar aircraft model and erroneous stability coefficients can result.
- Verify that the PTNN recalls proper aircraft model data and feeds it to DCS.
- Verify that DCS receives PTNN and PID data in the proper form.

#### **4.2.7. Documentation Verification**

Documentation verification makes certain that:

- Documentation is adequate, complete and consistent
- Documentation preparation is timely
- Configuration management of documents follows specified procedures

For the most part, descriptions of adaptive systems should be included in standard project documentation. For example, the Verification Plan should contain a plan to verify neural nets. Any special documentation about adaptive systems should be verified for technical accuracy by a peer review, then, catalogued and safeguarded following standard documentation configuration management procedures.

### 4.3. Validation of Neural Networks

Validation guidelines for Adaptive Systems are based on March 1998 IEEE/IEA 12207.0, paragraph 6.5. The validation process is documented in the validation plan and consists of the following:

- Items subject to validation
- Validation environment
- Testing:
  - Prepare test requirements, test cases and test specifications for analyzing test results
  - Ensure test requirements, test cases and test specifications reflect requirements
  - Conduct tests:
  - Validate that the software satisfies its intended use

Each of these steps will be discussed as they relate to validation of adaptive systems.

#### 4.3.1. Items Subject to Validation

Both PTNN and OLNN will be subject to validation to ensure they perform within acceptable ranges. Acceptable ranges criteria must be defined in the requirements and may include a range of error percentages, maximum number of cycles required to learn, time limits for learning, et al.

#### 4.3.2. Validation Environment

The validation environment includes everything necessary to conduct a test of the adaptive system including hardware, software and qualified test engineers with training and expertise on V&V techniques for neural nets.

##### Testbeds (Hardware)

Low, medium and high-fidelity testbeds will be necessary to properly test adaptive systems. Samples of these testbeds are described in the following table:

Testbed Name	Fidelity	Test Hardware
Aircraft	Highest	Flight
Simulator 2	Higher	Simulator with flight hardware and actual redundant flight control system
Simulator 1	Medium	Simulator with some flight hardware and models of flight control software
Batch Simulation	Lowest	Models running on typical Unix/Linux workstation with no flight hardware.

The Linear Simulator is the lowest fidelity and least expensive, testbed. It runs on a typical workstation and all aspects of the simulation are modeled in software including aerodynamics, engine models, and flight processors. The Linear Simulator is used to repeatedly run functional tests and regression tests.

In order to be cost-effective, the lowest fidelity testbed is used as much as possible. However, as the system becomes more complex, certain testing can be run only on higher fidelity testbeds. Before using a higher fidelity testbed, significant testing is done from a one-step lower fidelity testbed to ensure no errors at that level.

Higher fidelity testbeds like Simulators 1 and 2 contain a combination of software models and flight hardware. The most common elements of these testbeds are some of the flight processors, communication buses and a cockpit. These testbeds allows for additional testing not possible in a simpler configuration. Typically they contain software models of nonlinear aerodynamics, engine dynamics, actuator models, and sensor models. The hardware in the loop simulator includes the redundant flight control computers. This configuration allows for a complete check out of all interfaces to the flight hardware, processor timing tests, and various failure modes and effects (FMEA) testing.

The aircraft is what you could call the “ultimate simulator”. It maximizes the use of flight hardware components and minimizes the number of software models. This configuration can be accomplished in several different ways, the actual aircraft may be tied into the nonlinear simulation, or an iron-bird aircraft may be used to provide actuators, sensor noise, actual flight wiring, and some structural interactions. This final configuration can also be accomplished by placing flight hardware (sensors, and actuator, and other flight components interfaced to the nonlinear simulation) in a laboratory.

### Software

As shown in the following table, different versions of NN software are available for testing purposes.

NN	Software
PTNN	<ul style="list-style-type: none"> <li>• Ada</li> <li>• C running on Unix/Linux</li> <li>• C running on VxWorks</li> <li>• Matlab</li> <li>• Simulink</li> </ul>
DCS	C (contains no dynamic memory and no global data)

### 4.3.3. Testing

Testing is an exercise to validate that the software satisfies its intended use. Normally, test cases are developed to follow critical paths through the software and each test case has expected results. Because the PTNN has fixed weights, it can be tested via this traditional approach using the following advanced techniques:

- Perform Unit Testing. Neural nets used to be considered a black box, however, NASA devised a way to modularize NN code so white box testing can also be applied. Therefore, it is possible to perform unit testing on neural net code to find errors.

Unit testing may be requirements-driven or design-driven. Requirements-driven or black box testing is done by selecting the input data and other parameters based on the NN software requirements and observing the outputs and reactions of the software. Black box testing can be done at any level of integration. In addition to testing for satisfaction of requirements, some of the objectives of requirements-driven testing are to ascertain:

- Computational correctness.
- Proper handling of boundary conditions, including extreme inputs and conditions that cause extreme outputs.
- Proper behavior under stress (fixed and random inputs) or high load (many inputs at once). **Note:** *The aerospace industry also includes this task in the verification process.*
- Adequate error detection, handling, and recovery

Design-driven or white box testing is the process where the tester examines the internal workings of code. Design-driven testing is done by selecting the input data and other parameters based on

the internal logic paths to be checked. The goals of design-driven testing include ascertaining correctness of:

- All paths through the NN code. For most software products, this can be feasibly done only at the unit test level.
  - Bit-by-bit functioning of interfaces
  - Size and timing of critical elements of software code. **Note:** *The aerospace industry also includes this task in the verification process.*
- Test the frequency response and phase and gain margins - these tests require a model of the aircraft dynamics and are performed to ensure proper mil specs are met for stability
  - Perform failure modes and effects (FMEA) testing - failure modes and effects testing is one of the most challenging because of the various types and combinations of failures. For the PTNN, failures of all input parameters need to be tested. Tolerance "failures" as well as true failures need to be evaluated to ensure system performance is within design requirements specifications.
  - Sensitivity analysis - usually performed on flight control gains to ensure proper stability and flying qualities. These values can be used to determine the maximum error for the PTNN. Careful consideration should be given to the transonic area where gains can change rapidly and therefore must be more accurate.
  - Timing - flight computer timing tests must ensure that, under the worst case scenario, the system has adequate performance, or degrades gracefully when saturated. Currently, methods have been used to measure spare CPU time. If the PTNN is interfaced using communication protocols, such as mil-std 1553, timing tests are also required with the entire system operating as in flight.
  - System utilization - can refer to timing, but it also refers to memory available, data through put and other system specific resources.
  - Time to adapt for DCS and OLNN. *Note: PTNN do not adapt.*
  - Piloted evaluation - ability of the pilot to fly the aircraft in all conditions is paramount. Nominal, across the envelope, flying qualities must be evaluated, along with evaluations with failure modes and transients.

Additional validation activities for IFCS PTNN include:

- Checking interpolation of wind tunnel data for PTNN to check for gaps during training
- For PTNN, check points between data to make sure the curve is consistent (no unexpected jumps).
- List stability coefficients output from PTNN and compare against valid ranges
- Check that PTNN data sent to ground via telemetry is accurate to validate that the PTNN is working in the flight environment.
- Compare training values with actual output values

## 4.4. V &V Metrics

The following metrics used for traditional software may be applied to NN:

Testing Metrics:

- Planned and actual hours for each NN test
- Tracking of NN tests to NN requirements - % Complete
- Number of NN Tests (Cases/Procedures)
- Planned and actual start date for each NN test
- Planned and actual end date for each NN test
- Number of NN defects by classification
- Time required for NN to learn

Metrics for Inspections and/or Reviews:

- Planned and actual hours for each reviews (peer and independent)
- Date package distributed for review or inspection
- Planned and actual start date for each review
- Planned and actual end date for each review
- Date defects consolidated after review or inspection
- Base number of defects (critical vs. non-critical)

NN Metrics:

- Learning time
- Recall response time
- Accuracy
- Repeatability
- Robustness in the face of failures



## 4.5. Independent Verification and Validation (IV&V)

Independent Verification and Validation (IV&V) is a process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that is neither the developer nor the acquirer of the software.

*NASA Procedures and Guidelines (NPG) 8730.DRAFT 2, Software Independent Verification and Validation (IV&V) Management* discusses the requirements for independent verification and validation. In a nutshell, a manned mission and any mission or program costing more than \$100M will require IV&V.

The IV&V agent should have no stake in the success or failure of the software. The IV&V agent's only interest should be to make sure that the software is thoroughly tested against its complete set of requirements.

The IV&V activities duplicate the V&V activities step-by-step during the life cycle, with the exception that the IV&V agent does no informal testing. If there is an IV&V agent, the official acceptance testing may be done only once, by the IV&V agent. In this case, the development team will demonstrate that the software is ready for official acceptance.

## 5.APPENDIX A: ACRONYMS

Term	Definition
AI	Artificial Intelligence
ANSI	American National Standards Institute
ARC	Ames Research Center
CCB	Change Control Board
CM	Configuration Management
COTS	Commercial Off The Shelf
CVS	Concurrent Version System
EIA	Electronic Industries Association
FMEA	Failure Mode Effects Analysis
FMECA	Failure Mode Effects and Criticality Analysis
FIFO	First In First Out
IEC	International Electro-technical Commission
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Organization for Standardization
IV&V	(NASA) Independent Verification & Validation
MIL STD	Military Standard
NASA	National Aeronautical Space Administration
NPD	NASA Policy Directive
NPG	NASA Procedures and Guidelines
PID	Parameter Identification
RMA	Reliability, Maintainability, Availability
RTCA	Requirements and Technical Concepts for Aviation
SW	Software
UML	Unified Modeling Language
USA	United Space Alliance
V&V	Verification & Validation

**Note:** More Acronyms: <http://www.ksc.nasa.gov/facts/acronyms.html>

## 6.APPENDIX B: GLOSSARY

Items	Definition
Adaline	Adaptive element for multilayer feedforward networks introduced by Widrow
Algorithm	A rule or procedure for solving a problem <sup>17</sup>
ANN	Artificial Neural Network. Large parallel network of interconnected elements and their organization.
Axon	Output branch from a biological neuron consisting of many collateral branches that contact different neurons.
Back Propagation	Weight-vector optimization method used in multilayer feedforward networks. Differences in the in the actual outputs are measured with respect to the desired outputs are feed backwards towards the inputs and used to adjust the weights of the neurons.
Black Box testing	Requirements-driven testing where engineers select system input and observe system output/reactions
Codebook Vector	A parameter vector similar to a weight vector but used as a reference vector in vector quantization methods such as Self Organizing Maps and Learning Vector Quantization paradigms.
Connection	Link between neurons, coupling signals is proportion to its weight parameter.
CSCI	CSCI – Computer Software Configuration Item (a term used in NASA or Military standards to describe a product like a jet engine or a computer system.
Eigenvalue	Scalar value for which an operator equation of a linear equation can be determined.
Epoch	A finite set of meaningful inputs patterns introduced sequentially
Feedback Network	A network in which signal paths can return to the same node. Feedforward network A network in which signal paths can never return to the same signal node.
Feedforward Perceptrons	The term perceptron refers to a feedforward network of nodes with response like those shown in Figure 2 above. Feedforward means there are no connection loops that would allow output to feed back to their inputs and change the output at a later time. In other words, the network implements a static mapping that depends only on present inputs and is independent of previous system states.
Fidelity	Integrity of testbed. For example: low fidelity testbed may have a simulator rather than actual spacecraft hardware. The highest fidelity testbed is the actual hardware being tested
FMEA/FMECA	FMEA – Failure Mode Effects Analysis FMECA - Failure Mode Effects and Criticality Analysis FMEA and FMECA aides in determining what loss of functionality occurs due to an unremediated fault state
Gram-Schmidt	An algebraic method used to find orthogonal basis vectors in a linear subspace.
Hamming Distance	The number of dissimilar elements in two ordered sets.

Items	Definition
Hard-Limit Output	A result or output, usually associated with a neural transfer function, that becomes saturated when its value reaches an upper or lower bound.
Hebb's Law	The most frequently cited learning law in neural-network theory. The synaptic activity is assumed to increase in proportion to the product of presynaptic and postsynaptic stimulus.
Hidden Layer	An intermediate layer of neurons in a multilayer feedforward network that has no direct signal connection to inputs or outputs.
Hopfield Network	A state transition neural network that has feedback connections between all neurons. The energy-function paradigm was developed by Hopfield.
Inhibition	A synaptic control action that decreases the activation of a neuron by hyperbolizing its membrane.
Input Activation	An implicit variable associated with a neuron, usually an input signal or a synaptic weight. The output activity is determined as a function or differential equation related to the input activation.
Input Layer	The layer of neurons which receives input signals and stimulus.
Learning Rate	The true learning rate or rate of change in parameters relating to one learning step or time constant.
Learning Rate Factor	The factor used to multiply the error rate of a system parameter which determines the learning rate.
Learning Rate Quantization	A supervised-learning vector quantization method where the decision surfaces relating to the Bayesian classifier are defined by the nearest-neighbor classification with respect to sets of codebook vectors assigned to each class and describing it.
Lyapunov Function	A non-negative objective function that can be defined for optimization problems. If there exists a Lyapunov function, every learning step decreases it and a local minimum is reached.
Madaline	A multilayer feedforward network made of Adaline elements.
Markov Model	A statistical model that describes input-output relations of sequentially occurring signals using internal hidden states and transitional probabilities between them. The probability functions of the hidden states are identified on the basis of training data, usually in the training data.
Nominal	Expected behavior for no failure, for example: nominal behavior for a valve may be "open" or "shut"
Off-Nominal	Unexpected failure behavior, for example: off-nominal behavior for a valve may be "stuck open" or "stuck shut"
Software V&V	<p>Process of ensuring that software being developed or changed will satisfy functional and other requirements (verification) and each step in the process of building the software yields the right products (validation). In other words:</p> <ul style="list-style-type: none"> <li>• Verification – Build the Product Right</li> <li>• Validation – Build the Right Product</li> </ul>

Items	Definition
Telemetry	Process of measuring data at the source and transmitting it automatically. Telemetric applications include measuring and transmitting data from space flights, meteorological events, wildlife tracking, camera control robotics and oceanography studies. <sup>18</sup>
Validation	Build the Right Product
Verification	Build the Product Right
White Box Testing	Design-driven testing where engineers examine internal workings of code

## 7.APPENDIX C: FOR MORE INFORMATION

### **V&V Standards:**

IEEE Standard for Software Verification and Validation 1012-1998

### **Websites:**

MIL STD 498

[http://www.pogner.demon.co.uk/mil\\_498](http://www.pogner.demon.co.uk/mil_498)

<http://www.comsivam.org/reference/498/>

## 8.APPENDIX D: SELF ORGANIZING MAPS

The Self-Organizing Map (SOM) was developed by Prof. Teuvo Kohonen in the early 1980s as a visualization tool to aid in the evaluation and understanding of high- dimensional data. SOMs are represented as sheet-like neural-network arrays, which reduce and project highly dimensional data on a map of 1 or 2 dimensions and plot the similarities of data by grouping similar data items together. The goal is to construct topology-preserving mappings of training data where the location of a unit facilitates the clustering of data, creating a two-dimensional display of the input space that is easy to visualize.

The learning process is competitive and unsupervised, that is, no teacher is needed to define the correct output for an input. Only one map node (winner) at a time is activated corresponding to each input. Neurons are allowed to change themselves by learning to become more like samples in hopes of winning the next competition. The locations of the responses in the array tend to become ordered in the learning process. The basic premise in the SOM learning process is that, for each sample input vector, the winner and the nodes in its neighborhood are changed closer to the input vector in the input data space. SOMs organize themselves by competing for representation of the samples. Neurons are allowed to change themselves by learning to become more like input samples in hopes of winning the next competition. It is this selection and learning process that makes the weights organize themselves into a map representing similarities.

To construct a SOM, the weight vectors are first initialized. Then, a sample vector is randomly selected and the map is searched for weight vectors to find which weight best represents that sample. Each weight vector has a location and neighboring weights that are close to it. Being able to become more like that randomly selected sample vector rewards the weight that is chosen. In addition to this reward, being able to become more like the chosen sample vector also rewards the neighbors of that weight.

The selection of the **best matching unit** consists of inspecting all of the weight vectors and calculating the distance from each weight to the chosen sample vector. The weight with the shortest distance is the winner. If there is more than one with the same distance, then the winning weight is chosen randomly among the weights with the shortest distance. The most common method for determining that distance is to calculate the Euclidean distance, i.e.

$$\sqrt{\sum_{i=0}^n x_i^2}$$

where  $x[i]$  is the data value at the  $i$ th data member of a sample and  $n$  is the number of dimensions to the sample vectors. Scaling the neighboring weights: determining which weights are considered as neighbors and how much each weight can become more like the sample vector. The neighbors of a winning weight can be determined using a number of different methods.

The amount of neighbors decreases over time. This is done so samples can first move to an area where they will probably be, then they jockey for position. This process is similar to coarse adjustment followed by fine tuning.

The initial radius is set really high, some value near the width or height of the map.

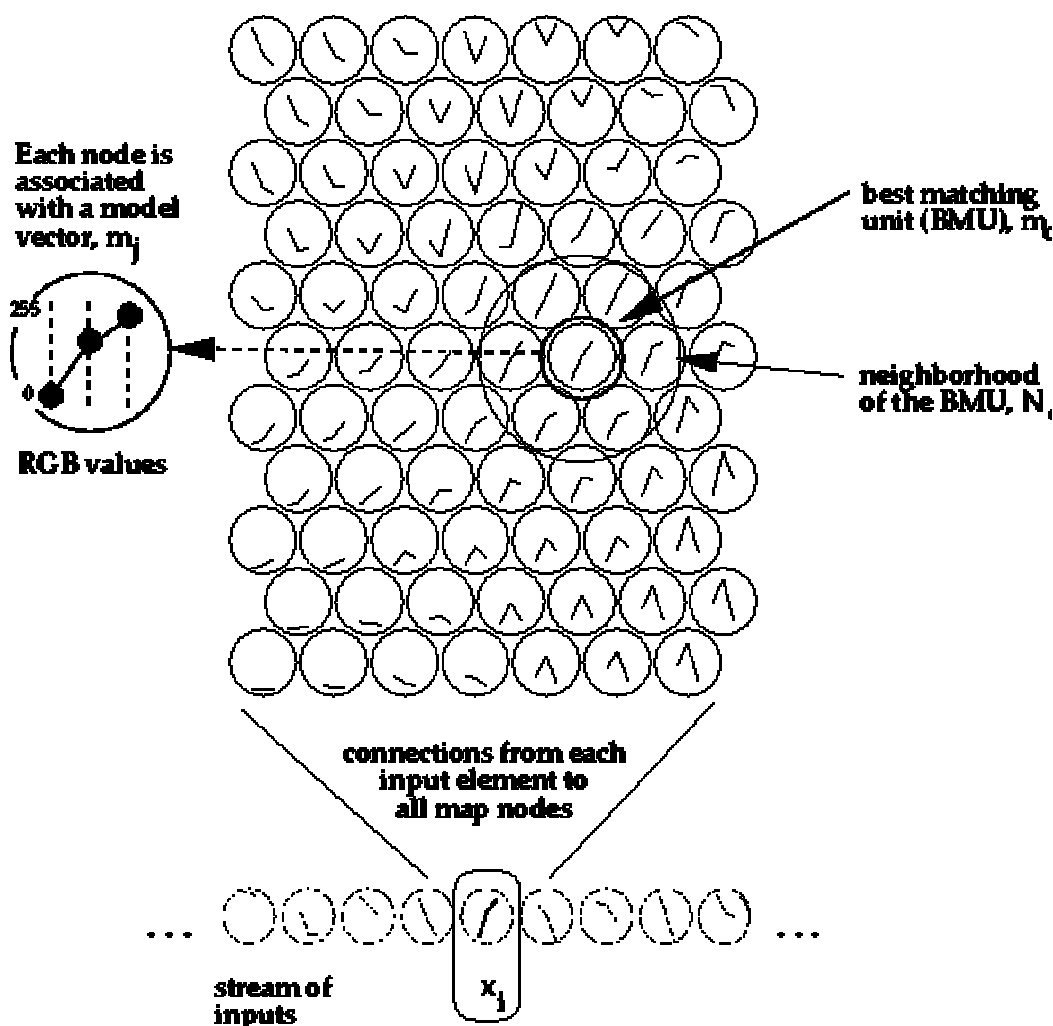
Sample data sets consisting of real vectors or the discrete-time coordinates are mapped onto an array. Each node  $i$  in the map contains a model vector, which has the same number of elements as the input vector.

The stochastic SOM algorithm performs a regression process. The initial values of the components of the model vector may be selected at random but are more efficiently initialized in some orderly fashion, e.g., along a two-dimensional subspace spanned by the two principal eigenvectors of the input data vectors.

Any input item is thought to be mapped into the location, the output of which matches best with in some metric. The self-organizing algorithm creates the ordered mapping as a repetition of the following basic tasks:

An input vector is compared with all the model vectors. The best-matching unit (node) on the map, i.e., the node where the model vector is closest to the input vector in some metric is identified. This best matching unit is called the winner.

2. The model vectors of the winner and a number of its neighboring nodes in the array are changed towards the input vector according to the learning principle specified below.





Adaptation of the model vectors in the learning process may take place according to the following equations:

$$\begin{aligned} \mathbf{m}_i(t+1) &= \mathbf{m}_i(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{m}_i(t)] && \text{for each } i \in N_c(t), \\ \mathbf{m}_i(t+1) &= \mathbf{m}_i(t) && \text{otherwise,} \end{aligned}$$

where  $t$  is the discrete-time index of the variables, the factor

$$\alpha(t) \in [0, 1]$$

is a scalar that defines the relative size of the learning step, and  $N_c(t)$  specifies the neighborhood around the winner in the map array.

At the beginning of the learning process the radius of the neighborhood is fairly large, but it is made to shrink during learning. This ensures that the global order is obtained already at the beginning, whereas towards the end, as the radius gets smaller, the local corrections of the model vectors in the map will be more specific. The factor also decreases during learning.

One method of evaluating the quality of the resulting map is to calculate the average quantization error

$$\{\|\mathbf{x} - \mathbf{m}_c(\mathbf{x})\|\}$$

over the input samples, defined as  $E$

$c$  indicates the best-matching unit for  $\mathbf{x}$ . After training, for each input sample vector the best-matching unit in the map is searched for, and the average of the respective quantization errors is returned.

## 9. APPENDIX E: SAMPLE REQUIREMENTS FOR THE PRE-TRAINED NEURAL NETWORK:

The PTNN shall [ ] generate sufficient stability derivatives to provide values used by other software on the F-15 research aircraft including the SOFFT controller for flight control. The PTNN outputs shall [ ] be accurate enough to ensure vehicle stability through out the flight envelop.

Compact – shall [ ] require less than 512KB of EPROM, 256K of RAM (storage available in the original computational environment, the VMSC)

Execute in real-time in an on-board aircraft processor – Execute at approximately 40Hz in the F-15 ACTIVE VMSC processor (a Motorola 88000 processor)

High accuracy over entire flight envelope – Maximum errors less than approximately 15% of variable range, rms. errors less than approximately 5% of variable range.

### Network Inputs

The inputs for the PTNN are mach, altitude, alpha (angle of attack), beta (sideslip angle), stabilator deflection, canard deflection, rudder deflection, and aileron deflection.

The maximum number of inputs any one sub network will have is 5. The minimum number of inputs for any sub network is 3.

Altitude dimensions are in feet. Alpha, beta, stabilator deflection, canard deflection, rudder deflection, and aileron deflection are in degrees. Mach is in mach number.

The inputs of beta and rudder deflection are magnitudes only and they are passed through an absolute value function to eliminate their sign.

Networks for  $C_{z_{\bar{D}_1}}$ ,  $C_{m_{\bar{D}_1}}$ ,  $C_{z_{\bar{D}_c}}$ ,  $C_{m_{\bar{D}_c}}$ ,  $C_{z_{\bar{D}_s}}$ ,  $C_{m_{\bar{D}_s}}$  have inputs of mach, altitude, alpha, stabilator deflection, and canard deflection.

Network for  $C_{z_{\bar{D}_2}}$  has inputs of mach, alpha, and beta.

Network for  $C_{m_{\bar{D}_2}}$  has inputs of mach, alpha, beta, and rudder deflection.

Network for  $C_{m_q}$  has inputs of mach, altitude, and alpha.

Networks for  $C_{y_{\bar{D}_1}}$ ,  $C_{l_{\bar{D}_1}}$ ,  $C_{n_{\bar{D}_1}}$  have inputs of mach, altitude, alpha, beta, and canard deflection.

Networks for  $C_{y_{\bar{D}_2}}$ ,  $C_{l_{\bar{D}_2}}$ ,  $C_{n_{\bar{D}_2}}$ ,  $C_{y_{\bar{D}_r}}$ ,  $C_{l_{\bar{D}_r}}$ ,  $C_{n_{\bar{D}_r}}$  have inputs of mach, altitude, alpha, beta, and rudder deflection.

Networks for  $C_{y_{\bar{D}_a}}$ ,  $C_{l_{\bar{D}_a}}$ ,  $C_{n_{\bar{D}_a}}$  have inputs of mach, altitude, alpha, and aileron deflection.

Networks for  $C_{y_{\bar{D}_4}}$ ,  $C_{l_{\bar{D}_4}}$ ,  $C_{n_{\bar{D}_4}}$  have inputs of mach, alpha, beta, and stabilator deflection.

Networks for  $C_{y_{\delta r}}$ ,  $C_{l_{\delta r}}$ ,  $C_{n_{\delta r}}$  have inputs of mach, altitude, alpha, and stabilator deflection.

Networks for  $C_{y_{\delta DC}}$ ,  $C_{l_{\delta DC}}$ ,  $C_{n_{\delta DC}}$  have inputs of mach, altitude, alpha, and canard deflection.

Networks for  $C_{l_p}$ ,  $C_{n_p}$ ,  $C_{l_r}$ ,  $C_{n_r}$  have inputs of mach, altitude, and alpha.

### Input Domain

Training data applied to the neural networks shall [ ] be generated from the existing ACTIVE database (95-96 Annual IFC Report). The PTNN was trained and tested for a small range of the F-15 ACTIVE's flight envelope. The envelope is defined as:

5000 feet < **ALTITUDE** < 50000 feet  
0.2 < **MACH** < 1.6

All of the control surface deflections shall [ ] be modeled for the training data to their respective position limits except for the positive canard deflection. The positive canard deflection was modeled to 5 degrees instead of the maximum limit of 15 degrees because the canard on the F-15 ACTIVE schedule does not exceed 5 degrees. Sideslip angle data was included for sideslips up to 10 degrees for all flight conditions.

Inside of the training flight envelope, two sub-envelopes were defined based upon alpha. Some of the networks experienced large training/testing errors or network sizes, so the flight envelope was reduced to avoid these regions. (No data exists for which networks have reduced flight envelopes.)

The result of the data selection from the ACTIVE database shall [ ] be 34 training sets of 3-5 independent variables with approximately 500-100,000 records each.

It is important to note that this envelope does not include a landing configuration or thrust vectoring effects.

### Input Pre-Processing

The eight inputs into the PTNN shall undergo some pre-processing before their inputs are applied to the network.

The absolute value of beta is re-assigned to the beta variable. If beta is larger than 10.0, it is assigned a value of 10.0.

The absolute value of the rudder deflection is re-assigned to the rudder variable. If the rudder deflection is larger than 30.0, it is assigned a value of 30.0.

If mach is smaller than a value of 0.2, it is re-assigned a value of 0.2. If mach is larger than 1.6, it is re-assigned a value of 1.6.

If altitude is smaller than 5000.0, it is re-assigned a value of 5000.0. If altitude is larger than 50000.0, it is re-assigned a value of 50000.0.

If the stabilator deflection is smaller than -30.0, it is re-assigned a value of -30.0. If the stabilator deflection is larger than 15.0, it is re-assigned a value of 15.0.

If the aileron deflection is smaller than -40.0, it is re-assigned a value of -40.0. If the aileron deflection is larger than 40.0, it is re-assigned a value of 40.0.

If the canard deflection is smaller than -35.0, it is re-assigned a value of -35.0. If the canard deflection is larger than 5.0, it is re-assigned a value of 5.0.

If alpha is smaller than  $-4.0$ , it is re-assigned a value of  $-4.0$ .

If mach is less than or equal to  $0.85$  or altitude is larger or equal to  $25000.0$  and only then if alpha is greater than  $20.0$ , alpha is re-assigned a value of  $20.0$ . If the mach and altitude conditions are not met and if alpha is greater than  $14.0$ , alpha is re-assigned to  $14.0$ .

### Network Topology

The network chosen for the PTNN shall [ ] be a multilayer perceptron network architecture. The perceptrons shall be able to accept multiple inputs. The perceptrons shall use a digital approximation to a hyperbolic tangent as the activation function for processing of input data.

Each network consists of at least one hidden layer of neurons and a possible second layer of neurons. The output of the sub-network is a weighted summation of the outputs from each neuron in the last layer which is combined with a bias. The weights are adjusted based on the learning algorithm. Once trained, the weights are frozen and shall not change during operational use.

### Network Outputs

There shall [ ] be a total of 44 individual values which are generated as outputs for the PTNN. They are as follows:

$$C_{z_{D1}} \quad C_{z_{D2}} \quad C_{z_{Dcf}} \quad C_{z_{Dcs}} \quad C_{z_{Dc}} \quad C_{z_{Ds}}$$

$$C_{m_{D1}} \quad C_{m_{D2}} \quad C_{m_{Dcf}} \quad C_{m_{Dcs}} \quad C_{m_{Dc}} \quad C_{m_{Ds}} \quad C_{m_q}$$

$$C_{y_{D1}} \quad C_{y_{D2}} \quad C_{y_{D4}} \quad C_{y_{D}} \quad C_{y_{Dr}} \quad C_{y_{DDC}} \quad C_{y_{Dr}} \quad C_{y_{Da}}$$

$$C_{l_{D1}} \quad C_{l_{D2}} \quad C_{l_{D4}} \quad C_{l_{D}} \quad C_{l_{Dr}} \quad C_{l_{DDC}} \quad C_{l_{Dr}} \quad C_{l_{Da}} \quad C_{l_p} \quad C_{l_r}$$

$$C_{n_{D1}} \quad C_{n_{D2}} \quad C_{n_{D4}} \quad C_{n_{D}} \quad C_{n_{Dr}} \quad C_{n_{DDC}} \quad C_{n_{Dr}} \quad C_{n_{Da}} \quad C_{n_p} \quad C_{n_r}$$

$$\frac{\square c}{\square a}, \text{ commanded canard deflection, commanded stabilator deflection}$$

### Neural network learning method

Levenberg-Marquardt Optimization shall [ ] be used as a nonlinear optimization because it significantly outperforms gradient descent and conjugate gradient methods for medium sized problems. It is a pseudo-second order method which means that it works with only function evaluations and gradient information but it estimates the Hessian matrix using the sum of outerproducts of the gradients. The mean squared error of the difference from the neural network and the desired value shall [ ] be appropriate for the derivative range.

### Network verification

Once the network has been trained, it shall [ ] be tested using a different data set from that used for training. It shall [ ] cover the entire range values for each input into the network. The maximum error in any derivative shall be 15% and the average error less than 5%.

### System validation with the PTNN.

The following validation shall [ ] be performed:

- System Integration Test (Open-Loop with Flight Hardware)
- Pilot-Vehicle Interface, Mode Transitions
- Failure Modes, Timing
- Hardware-in-the-Loop Simulation
- Handling Qualities
- Pilot-Vehicle Interface, Mode Transitions
- Failure Modes, Timing

#### Gain sensitivity analysis

- Analyzed gain sensitivity to noisy inputs and sensor errors
- Error perturbations on nominal Neural Net inputs

Two flight conditions shall [ ] be examined  
0.64M/16kft, 1.35M/34kft

All 8 neural net inputs to be perturbed

AOA	$\pm 2^\circ$ in $0.25^\circ$ increments
Beta	$\pm 2^\circ$ in $0.25^\circ$ increments
Mach	$\pm 0.04$ Mach in $0.005$ M increments
Altitude	$\pm 500$ ft in $125$ ft increments
Collective Stab	$\pm 4^\circ$ in $0.5^\circ$ increments
Collective Canard	$\pm 4^\circ$ in $0.5^\circ$ increments
Rudder	$\pm 4^\circ$ in $0.5^\circ$ increments
Differential Aileron	$\pm 4^\circ$ in $0.5^\circ$ increments

Maximum within tolerance failures simulated on all single and dual string sensors

Sensors analyzed:

$\alpha$  (single string)

AOA probes (dual string)

Qc (dual string)

Ps (dual string)

Aero increments study shall [ ] be performed;

Three primary stability and control derivatives analyzed

$C_{m_\alpha}$ ,  $C_{m_{\alpha_{stab}}}$ ,  $C_{n_\alpha}$

Each derivative was incremented +100% and -50%

In all cases the flying qualities shall be acceptable, and flight control phase and gain margins within limits.

## 10. APPENDIX F: HESSIAN MATRIX

A Hessian matrix,  $H$ , of error with respect to the weights is the matrix of second derivatives with the following elements:

$$h_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$$

Knowledge of the Hessian is important for the following reasons:

- Convergence of many optimization algorithms is governed by characteristics of the Hessian matrix. In second-order optimization methods, the matrix is used explicitly to calculate search directions. In other cases, it may have an implicit role.
- Some pruning algorithms use Hessian information to decide which weights to remove
- The inverse Hessian can be used to calculate confidence intervals for the network outputs
- Hessian information can be used to calculate regularization parameters
- Hessian information can be used for fast retraining of an existing network when additional training data becomes available
- At a local minimum of the error function, the Hessian will be positive definite. This provides a way to determine if learning has stopped because the network reached a true minimum or because it “ran out of gas” on a flat spot<sup>19</sup>

# 11. APPENDIX F: STABILITY COEFFICIENTS

Stability Derivative Name	Stability Derivative Symbol
Coefficient of normal force with respect to alpha	$C_{z_\alpha}$
Coefficient of normal force with respect to canard deflection	$C_{z_{\alpha_c}}$
Coefficient of normal force with respect to stabilator deflection	$C_{z_{\alpha_s}}$
Coefficient of pitching moment with respect to alpha	$C_{m_\alpha}$
Coefficient of pitching moment with respect to canard deflection	$C_{m_{\alpha_c}}$
Coefficient of pitching moment with respect to stabilator deflection	$C_{m_{\alpha_s}}$
Coefficient of pitching moment with respect to pitch rate	$C_{m_q}$
Coefficient of side force with respect to sideslip	$C_{y_\beta}$
Coefficient of side force with respect to differential tail deflection	$C_{y_{\beta r}}$
Coefficient of side force with respect to differential canard deflection	$C_{y_{\beta DC}}$
Coefficient of side force with respect to rudder deflection	$C_{y_{\beta r}}$
Coefficient of side force with respect to aileron deflection	$C_{y_{\beta a}}$
Coefficient of rolling moment with respect to sideslip	$C_{l_\beta}$
Coefficient of rolling moment with respect to differential tail deflection	$C_{l_{\beta r}}$
Coefficient of rolling moment with respect to differential canard deflection	$C_{l_{\beta DC}}$
Coefficient of rolling moment with respect to rudder deflection	$C_{l_{\beta r}}$
Coefficient of rolling moment with respect to aileron deflection	$C_{l_{\beta a}}$
Coefficient of rolling moment with respect to roll rate (roll damping)	$C_{l_p}$
Coefficient of rolling moment with respect to yaw rate	$C_{l_r}$
Coefficient of yawing moment with respect to sideslip	$C_{n_\beta}$
Coefficient of yawing moment with respect to differential tail deflection	$C_{n_{\beta r}}$
Coefficient of yawing moment with respect to differential canard deflection	$C_{n_{\beta DC}}$
Coefficient of yawing moment with respect to rudder deflection	$C_{n_{\beta r}}$
Coefficient of yawing moment with respect to aileron deflection	$C_{n_{\beta a}}$
Coefficient of yawing moment with respect to roll rate	$C_{n_p}$
Coefficient of yawing moment with respect to yaw rate (yaw damping)	$C_{n_r}$

## 12. APPENDIX G: INTELLIGENT FLIGHT CONTROL SYSTEM (IFCS)

The Intelligent Flight Control System (IFCS) is being constructed as part of the Intelligent Flight Control (IFC) project, a collaborative effort among NASA Dryden Flight Research Center (DFRC), the NASA Ames Research Center (ARC); the Boeing Phantom Works (BPW); Institute for Software Research, Inc (ISR); and West Virginia University (WVU). Results from the IFC project will feed an overall strategy aimed at advancing flight control technology for civil and military aircraft, reusable launch vehicles, uninhabited vehicles, and space vehicles.

The goal of IFC is to develop and demonstrate a flight control system that can efficiently identify aircraft stability and control characteristics using neural networks and utilize this information to optimize aircraft performance in both normal and simulated failure conditions.

The IFCS will be tested in flight on the NASA F-15B (AFSN 71-0290 NASA 837). This aircraft has been highly modified from a standard F-15 configuration to include canard control surfaces, thrust vectoring nozzles, and a digital fly-by-wire flight control system. This aircraft has been previously used for intelligent flight controls work. Flight-testing the online learning system will demonstrate a flight control mode for a damaged fighter or transport aircraft that can return the aircraft safely to base.

### Background

During the last 30 years, at least 10 aircraft have experienced major flight control system failures claiming more than 1100 lives. Therefore, the National Transportation Safety Board (NTSB) recommended *“research and development of backup flight control systems for newly certified wide-body airplanes that utilize an alternate source of motive power separate from that source used for the conventional control system.”* NASA investigated and found that neural networks safely and effectively supply these alternate sources of power while providing consistent handling qualities across flight conditions and for different aircraft configurations. Under normal operating conditions, the flight control system uses conventional flight control surfaces. Under damage or failure conditions, neural nets allow the system to use unconventional flight control surface allocations, along with integrated propulsion control, when additional control power is necessary for achieving desired flight.

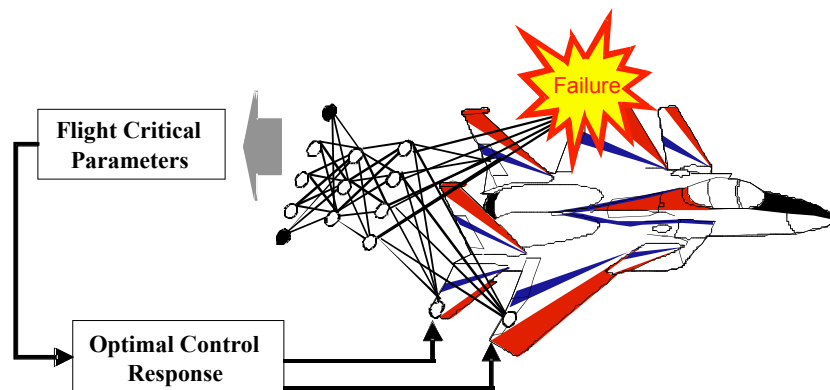
Additionally, neural networks allow the system to operate without emergency or backup flight control mode operations. This system can also utilize, but does not require, fault detection and isolation information or explicit parameter identification. In simulated flight tests by NASA test pilots, intelligent flight control systems improved handling qualities and significantly increased survivability rates under various simulated failure conditions.<sup>20</sup>

The IFCS incorporates the neural network technology from the NTSB research project described above.

### IFCS Example

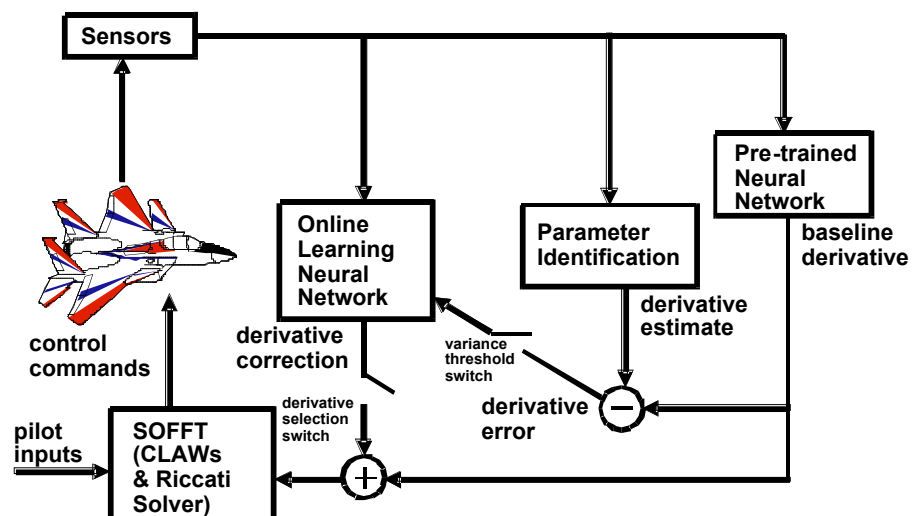
The following figure shows an overview of how the IFCS responds to a failure of a control surface (left aileron). Flight critical parameters are passed from sensors to a real-time parameter identification (PID) algorithm to measure the stability parameters of the aircraft (during flight). The onboard neural networks (PTNN and DCS) interpret the results from the PID and pass them to flight control system for use by adaptive control laws to optimize the flight response of the aircraft during this failure (as well as under a variety of maneuvering conditions).





**Figure 11: Failure of Control Surface and Optimal Control Response**

The subsequent figure illustrates a more detailed view of the IFCS. Sensor data flows to the Pre-Trained Neural Network (PTNN), Parameter Identification (PID) and Online Learning Neural Network (OLNN). The PTNN contains baseline derivatives computed from wind tunnel data. Because actual flight conditions may vary from wind tunnel data, actual sensor data from the PID is compared to the baseline derivative. Negative comparison results indicate a derivative error. This derivative error is passed to the OLNN for a correction. The OLNN computes a correction and passes it to the SOFFT controller, which takes into account the pilot inputs and provides control commands to the aircraft.



**Figure 12: IFCS Overview**

SOFFT (Stochastic Optimal Feed-Forward & Feedback Technology) is a flight control architecture that is based on an explicit model-following concept. The SOFFT controller attempts to match the desired performance characteristics, which are pre-selected by the pilot through the use of DAG (Dial a Gain) sets. It uses the neural network stability and control derivatives data to establish the plant model that controls the aircraft so it can achieve the desired handling qualities, and to continually optimize the feedback controller by integrating the neural network data in a real-time Riccati solver process that calculates feedback gains at 10 Hertz. CLAWS and the Riccati Solver are part of SOFFT.

# 13. APPENDIX H: V&V ISSUES for NEURAL NETWORKS

Copied directly from *V&V Issues for Neural Networks* by Johann Schumann







## 13.1. Introduction

V&V of Neural Networks is a very difficult topic. Even, for pre-trained neural networks (PTNN), no standard methods for their verification and validation have been developed. Neural network training algorithms are a specific kind of numerical optimization algorithms. Therefore, their properties and implementation needs to be carefully studied from the point of view of numerical code.

This report focuses in these latter aspects. It discusses a number of issues and criteria which are important for proper functioning of a neural-network based system. They all deal with numerical aspects of the training and network evaluation algorithms. Although this report is concentrated on PTNNs, many of these issues also arise for on-line trained NNs.

The aim of this report is to make the designer and the V&V person aware of (numerically oriented) problems which can occur in the design and implementation of a neural-network based system. Because, the issues, discussed in this report are often treated as implicit assumptions, it is worthwhile to check during the V&V phase if all these assumptions are indeed correct and have been implemented appropriately. Thus, this report tries to provide starting points and ideas for developing V&V processes (or to improve them), rather than to solve existing problems.

This report is structured as follows: In Section 12.2, we define the notion used throughout this paper. In the subsequent sections, we discuss individual topics:

- Section 12.3 covers problems which can occur with respect to the ranges of the inputs and outputs of the neural network, especially boundary values, or isolated singularities.
- Section 12.4 focuses on roundoff errors which can occur whenever real numbers (from ) are represented in a digital computer by floating-point numbers (e.g., `float` or `double`). Here, we discuss how round-off errors can influence the behavior of the neural-network training algorithms, and point out that for V&V purposes, it is important to also check the (built-in) library functions (e.g.,  or ) for accuracy.
- Input or output values which are magnitudes apart can lead to severe problems. Section 12.5 illustrates scaling problems and discusses the influence of (bad) scaling on the neural network recall and training behavior. We also demonstrate with an example that a seemingly simple function () can lead to very bad results when scaling is not done properly. In this section, we also describe techniques how to scale the input data and/or the NN training algorithm.
- An important characteristic of the approximated function is its smoothness and its behavior with respect to small changes in the input values. If, for a small change in the input, e.g., , the output differs substantially from the output, given , problems are close at hand. Section 12.6 gives an introduction into the sensitivity analysis of a neural network and illustrates the approach with two examples.
- A traditional metric for the behavior of a numerical problem is the so-called *condition number*. Section 12.7 discusses, how a condition number can be calculated for feed-forward neural networks, and it also describes some techniques for a fast, approximate calculation of the condition number.

- Section 12.8 focuses on numerical issues of the training algorithm. Here, we do not focus on a specific training algorithm (e.g., gradient descent or Levenberg-Marquardt). Rather, we consider the training problem in a generic way as the task to find a (global) minimum of the error function.

In this section, we first discuss the general properties of such an algorithm. Then, we illustrate with a well-known example, what can go wrong during the training: the progress of the training can go toward zero, i.e., we stall, or--despite good progress--we end up with a diverging problem.

Finally, we will focus on the termination criteria for a training algorithm: when to stop and how to stop. For proper operation of the training algorithms, a number of important considerations need to be taken into account.

Most of these topics should not be analyzed in isolation. So, for example, a problem with improper data ranges usually exhibits poor scaling, and thus can result in large roundoff errors and a poor condition number. On the other hand, a somewhat "complete coverage" of the numerically oriented V&V aspects of PTNNs requires to look at all these issues (from the different angles) such that no possible source of problems remains undetected.

Appendix I contains basic definitions about functions, quadratic forms, and matrices. Appendix J explains quadratic functions. Appendix K describes eigenvectors and eigenvalues. Appendix L contains derivatives of NN Activation functions.

## 13.2. Notation

In the following, we assume a simple network structure: a feed-forward network with one hidden layer.

Such a neural network architecture is shown in Figure 13. The network consists of  $n_{in}$  input nodes, i.e.,

an input value is a vector  $\mathbf{x}$  from  $\mathbb{R}^{n_{in}}$ . The network has  $n_{hid}$  hidden nodes which are connected to all input nodes and all  $n_{out}$  output nodes. The output function of each hidden node

is defined as:

$$f_i(\mathbf{x}) = \sum_{j=1}^{n_{in}} w_{ij} x_j + b_i$$

is defined as:

$$f_i(\mathbf{x}) = \sum_{j=1}^{n_{in}} w_{ij} x_j + b_i$$

$$f_i(\mathbf{x}) = \sum_{j=1}^{n_{in}} w_{ij} x_j + b_i$$
(1)

with weights  $w_{ij}$  and the *activation function*  $\sigma$ . The output of the neural network (at output node  $k$ ) is

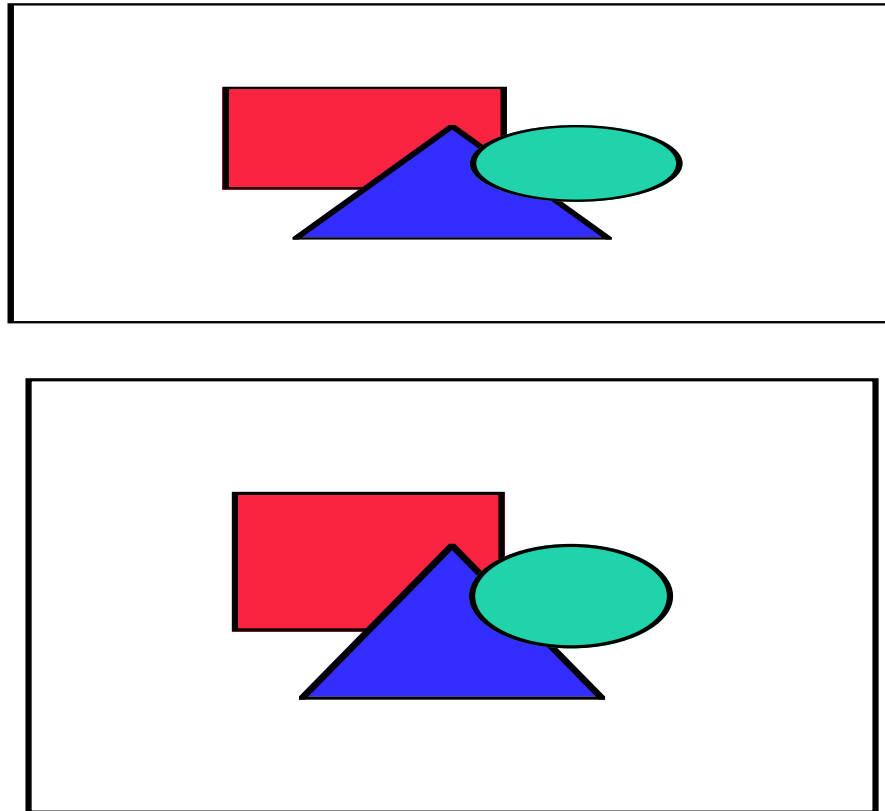
written as a weighted linear combination  $\sum_{i=1}^{n_{hid}} w_{ki} f_i(\mathbf{x}) + b_k$  of the output values of the nodes in the hidden layer

with  $\mathbf{w}_{out}$  that is


$$y_k = \sum_{i=1}^{n_{hid}} w_{ki} f_i(\mathbf{x}) + b_k$$

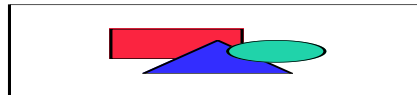
$$y_k = \sum_{i=1}^{n_{hid}} w_{ki} f_i(\mathbf{x}) + b_k$$
(2)

For purposes of compact notation, we combine all weights into a single matrix  $\mathbf{W}$ , and don't distinguish between the different layers. In more detail,  $\mathbf{W}$  is defined as




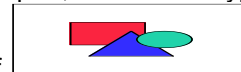
**Figure 13: Simple feed-forward network with a hidden layer (Figure produced by the tool SNNS <sup>21</sup>**

As the *activation function*  we (usually) use the *hyperbolic tangent*



A graph of this activation function is shown in Figure 14. In general, a variety of different functions can be used for Neural Networks. In order to allow the approximation of an arbitrary function (i.e., the network can in principle learn any function), the activation function needs to be non-linear. For better mathematical treatment, a smooth function with existing derivatives is preferred. In this report, we use the hyperbolic

tangent, because it is symmetric with respect to  with an output range of



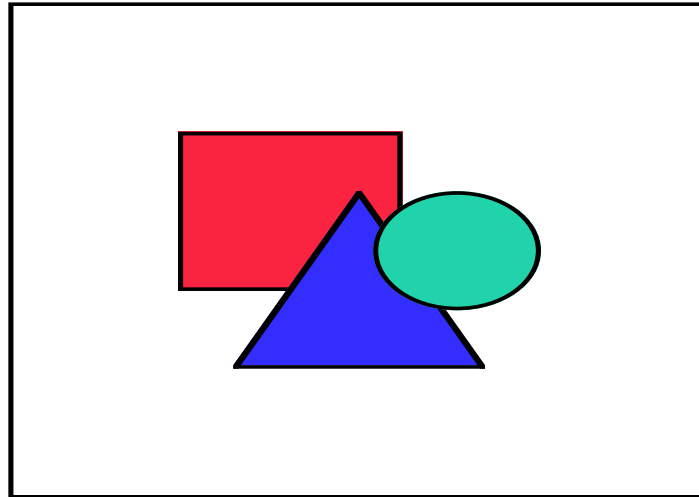
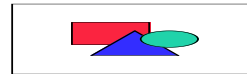


Figure 14: Non-linear activation function:




For the training, we assume that we have a set of training data which are pairs of input vectors






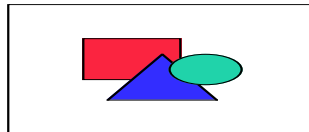
and corresponding desired (target) outputs



. When given as a set, e.g., , we use superscripts for indexing (not to be confused with exponentials). Thus, we have







Training of the neural network is the task of adapting the weights  and  in such a way that the error , defined as





is minimized. For performing this minimization (called *supervised learning*), a large number of different algorithms exist (e.g., gradient descent or Levenberg Marquardt).




For our discussion, we distinguish two phases:

1. *Training Phase*. During the training phase, the NN is presented with a training set . With a learning algorithm, we obtain the weight matrix , such that the error  is minimized.
2. *Deployment Phase*. After training of a pre-trained Neural Network (PTNN), all weights are frozen. This means, that during deployment, a fixed, read-only version of the weight matrix  is used in the system. Only in an on-line trained (or adaptive) NN architecture, a learning algorithm is running during deployment which can change the weights.

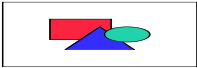


In a system with a PTNN, the desired network function is just obtained by evaluating the equations (1) and (2).

### 13.3. Data Ranges

In general, inputs and outputs for a NN are defined over , and , respectively. In practice, however, inputs and outputs have a specific meaning, and are thus restricted. These restrictions need to be enforced both for inputs and for outputs of the NN.


**Example** For a flight-control system, inputs for the neural net include , the Mach number which is naturally restricted to a range between , the elevation (in feet), and the angle of attack . The outputs of the NN comprise values, or derivatives.

**Example** Restrictions on the range of inputs and outputs and consistency of physical units need to be considered and tested carefully, as is briefly discussed in the following examples.

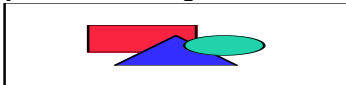


- A restriction of elevation to  can be dangerous in cases the aircraft needs to operate in Death Valley or other areas below sea level.
- Certain input values can lead to singularities. For example, angles of  are typical candidates, because any expressions containing tangents or cosines in the denominator lead to undefined values.
- Consistency of units and scaled units (e.g., km,  m) needs to be checked throughout the entire system. The classical example for a failure due to inconsistent units is the Mars Climate Orbiter<sup>22</sup>.



Ranges of input and output values also are closely related to scaling (Section 12.5) and sensitivity analysis (Section 12.6).


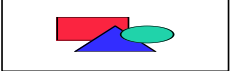


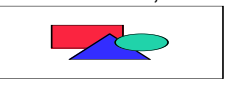

### 13.4. Roundoff Errors








The mathematical definition of the neural networks and the training algorithm assumes that all values are real numbers, taken from . In any practical implementation on a digital computer, however, these numbers must be represented with a finite number of binary bits. Most digital computers represent their numbers according to an IEEE standard. Because real numbers are represented as a pair, consisting of a mantissa and an exponent, these representations are called *floating-point numbers*. There is a wealth of literature (e.g., [23]), discussing these number representations.

Here, we will just focus some topics which are important with respect to neural networks. Round-off errors can be particularly large in expressions consisting of division and subtraction. Thus, in principle, each point in the algorithm where these two operators occur need to be analyzed with respect to round-off errors. Furthermore, round-off errors can propagate through an iterative algorithm. In the worst case, an initially small error can increase during each iteration loop, leading to divergence and bad results.

**Example** The training of a neural network is based upon the iterative minimization of the mean-square error . Here, we just consider the simple case with one output node (i.e., ). If, the values of  are very large and the training of the network proceeds, the difference

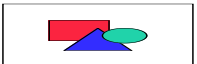
 can exhibit substantial round-off errors which lead to cancellation. In that case, an error of  would be the result (which would immediately terminate the learning algorithm).

So, for example for  and , the difference would still be  on a computer with double accuracy. For  and , this difference already results in .

**Example** At each step of the iterative training algorithm, a search direction  and a length  for this step is calculated. Then, the algorithm moves its search from  to . For convergence,  should always point towards the minimum, more specifically . Round-off errors, however, can result in situations where  suddenly points into a different direction. If this situation is not handled properly by the learning algorithm, the algorithm can start oscillating or diverging.

This problem can be extremely severe for neural networks with on-line learning.

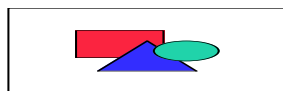
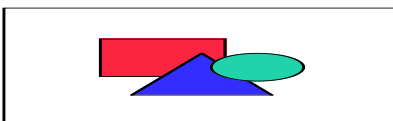
### 13.4.1. Accuracy of Operators

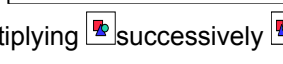





Built-in (library) operators (like ) need to be checked for their accuracy over the entire range of the floating-point numbers.

**Example** This problem actually happened to me recently when the author tried a floating-point library on a small embedded system (Lego RCX brick with a JVM).

Functions can be approximated by an expansion into a Taylor series (usually around 0). So the

approximation  can be written as:



For the exponential we get: . If this function is evaluated for larger ,  (which is usually calculated by multiplying  successively  times) produces excessive large numbers which result in very poor accuracy when divided by the (large) . Figure 15 shows the approximation error (squared error) between the approximated function and the Unix built-in exponential function (taken as "ground truth" here).



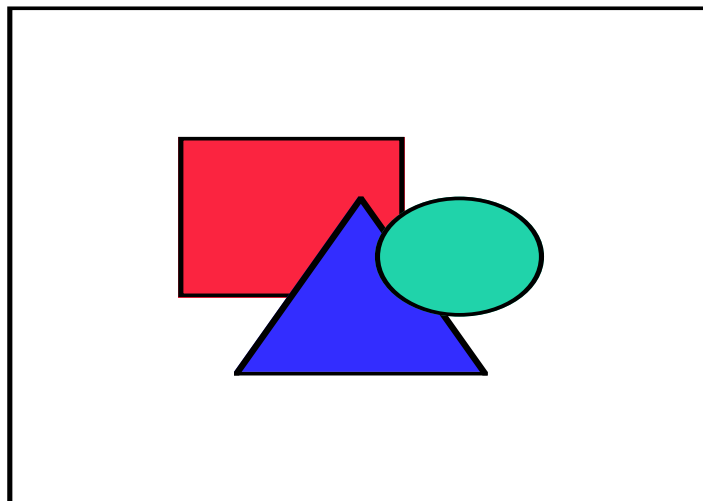







Figure 15: Square error between two different approximations of the  exponential function (logarithmic scale)

## 13.5. Scaling

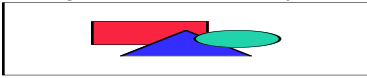
### 13.5.1. Badly Scaled Problems


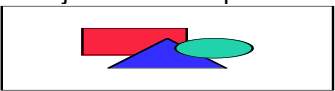
A scaling problem can occur when the values of different elements of the input vector are magnitudes apart. Then it is often the case, that the smaller values (although possibly crucial for calculation of the output) is not taken into account. This can lead to severe problems.

**Example** For the IFCS, some of the inputs have different orders of magnitude. For example, the elevation is in , whereas the Mach number might be between 0 and 5. An angle (in rad) might be even one or two orders of magnitude smaller (e.g.,  which is  rads). 

**Example** Likewise for navigation problems, the combination of measured distances (e.g., DME or GPS) and (possibly small) angles, e.g., VOR, can lead to similar situations which require scaling to work accordingly. 

**Example** One of the outputs of a DCFS PTNN has output values in the range of

. A naive training of a feed-forward network with the system's default values (stop when SSE is smaller than 0.001) yielded "learning success" after only 100 iterations. A closer look, however revealed that the "trained" neural network just set all outputs to zero. Due to the


small output values of , the error of the "trained" NN  turned out to be smaller than the built-in threshold. Therefore, training stopped almost immediately with "success".


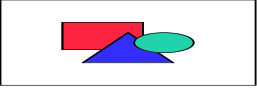
### 13.5.2. Influence of Scaling

Ideally, scaling should exhibit no influence on the system's behavior and the training. Rather it should avoid the problems discussed above. However, there are a number of intricacies which will be described in the following paragraphs.


### 13.5.2.1. Scaling and the Algorithm

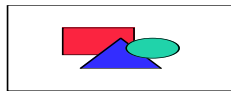
When scaling is performed on inputs and outputs, it is implicitly assumed that the algorithm in between (in our case the NN and the training algorithms) is not affected by this (except for round-off errors). However, this is not always the case, as the following example demonstrates.

**Example** Let us consider two kinds of learning algorithms: (a) gradient descent, and (b) a Newton method. Let us furthermore assume that we have a scaling matrix  such that the scaled input is

 and the scaled function is . If we look at the direction of the search step for each of the algorithms we can make the following observations.


(a)

The search direction  is always in the direction of the steepest descent, i.e.,





A little calculation reveals that change of units in various directions can affect the direction of the steepest descent. This effect shows up, if the scaling factors for different directions are not the same.

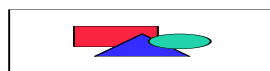
(b)

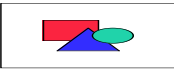

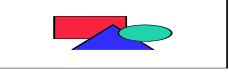

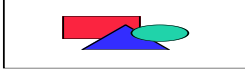
The search direction  for the Newton method always points to the lowest point of a quadratic model. Thus, whenever the scaling transformation moves the location of the lowest point, the direction will be adjusted automatically. Thus, the Newton method is stable with respect to scaling.



### 13.5.2.2. Scaling and Training

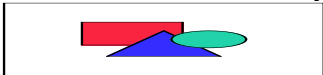
**Example** Let us consider the following example (cf. [24]): we want to learn the linear identity function


. For this simple task, we choose a small feed-forward network with one hidden layer with one node. This hidden node has an  activation function, the output node is linear. The entire transfer function can be written down as











with weights . Let us now consider two sets of training data: set 1 () is taken from the interval , set 2 () is taken from the interval .

Now, take a neural-network simulator and try to train the neural network with  and . As the desired

error we require  in our given interval. Especially for the training set taken from the large interval, convergence is extremely slow (if there is convergence at all). Despite the simplicity of the setting, the NN's behavior is far from good.

The following analysis of the problem shows, why. First, let us have a look at how the weights  have to be set to obtain the desired low residual error. It is obvious that, in order to approximate the

function  with a derivative of 1 must be mapped to a region where the activation function  also has a derivative close to 1. As is clear from Figure 14, this must be close to 0 (the derivative of  is 1 only at ). Thus,  must have a small value (to bring the argument of  close to 0), whereas the value of  must be large (of the order of ).


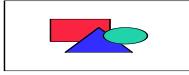






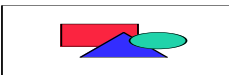
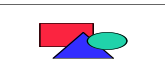
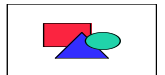

For our given training sets and the required error of , we roughly obtain the following values of the weights as shown in Table 1. The error  for  is shown in Figure 16. Note, that these values can be obtained analytically and are thus independent of the learning procedure (and its convergence behavior).

Table 1: Weight settings for learning the function .

Range			
	0.1	10	
			

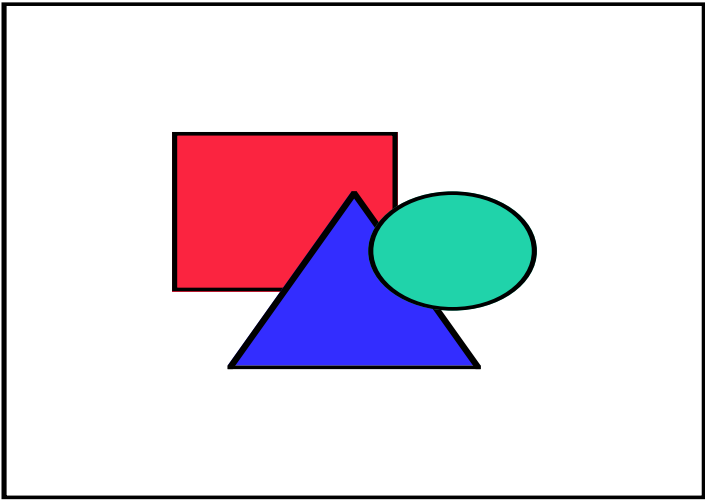

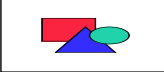

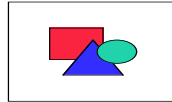


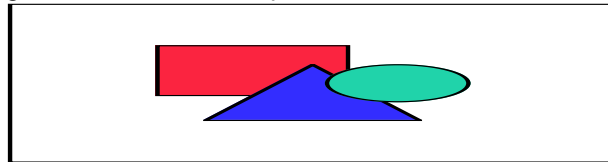
Figure 16: Error between output and exact value (o-x)

It is obvious that we have a scaling problem here. Multiplying the input value (which also could be quite small even for ) with such a small weight like  is calling for trouble.

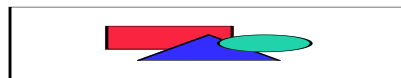
A more detailed *conditioning* analysis (see Section 12.7) reveals even more details. The condition number  is defined as the ratio of the absolute values of the largest and the smallest eigenvalue of the problem, i.e.,



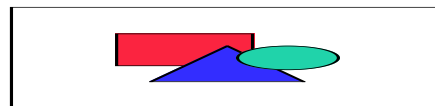
In our case, the conditioning is calculated with respect to the Hessian



with the quadratic error function




. A little calculation [25] reveals, for example, that

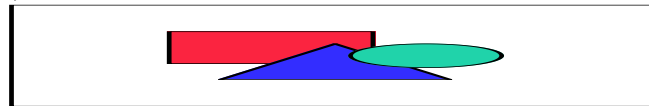


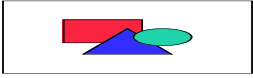
which results in



, which is a relatively bad condition number. The corresponding

matrix for  looks much, much worse:




yielding  which is a clear indication of a very ill-conditioned system.

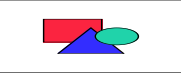
This example shows that seemingly smooth and simple target functions can exhibit severe problems in training.

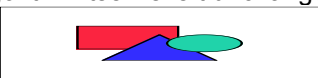
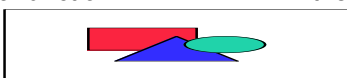

### 13.5.3. How to Scale

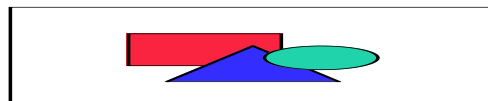
Scaling can be accomplished in two different ways: scaling of the input and output values, or scaling of the algorithm itself.

#### 13.5.3.1. Scaling the Input and Output

Scaling of the input and output values are accomplished by multiplication with a diagonal matrix . The

algorithm itself is left unchanged. For example, if for a function  the range of the first input

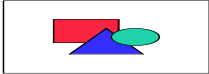

is , and for the second, , then a matrix  which scales everything into a 0-1 interval would be defined as:

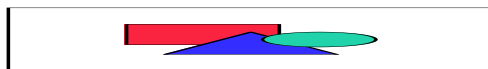


### 13.5.3.2. Scaling the Algorithm

Here, the appropriate steps inside the algorithm are adjusted appropriately.


**Example** For the gradient descent, we obtain the following modifications: the check for closeness to the

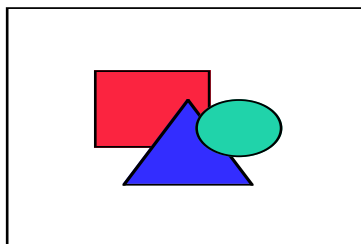
local minimum, usually  is replaced by  , and the search direction becomes:




### 13.5.3.3. Scaling and Biasing

Often scaling is combined with a linear transformation to remove a bias from the data. One common

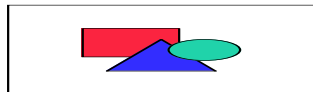
method for this kind of scaling is to linearly transform the input vectors  to have zero-mean and a standard deviation of one. This is accomplished by:








## 13.6. Sensitivity Analysis

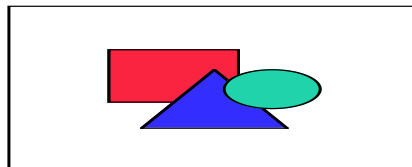
For a NN-based system it is important to know, how small changes in the input values  influence the output. This influence is called *sensitivity*. In a safety-critical application, care must be taken that the sensitivity of the system is limited, because high sensitivity can easily lead to unstable (or at least rough and bumpy) behavior.


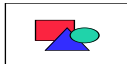
For a one-dimensional function  , we can define sensitivity  as the ratio




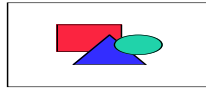
When we have  , the sensitivity is nothing but the derivative of the function  at point  . This is intuitively clear: whenever a function is very steep, small changes in  $x$  lead to drastic changes in  $y$ .





A measure of sensitivity can be similarly defined in the general case ( inputs  $()$  and  outputs  $()$ ). Here, we consider how a change in each input affects each output. Thus, we end up with the following matrix




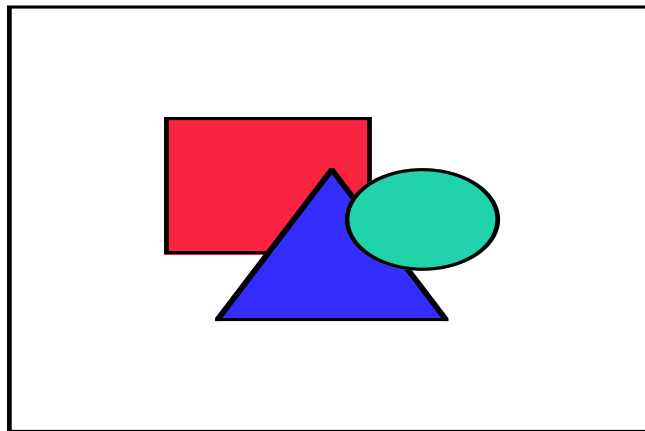
This matrix is called the *Jacobian* at  . In general, a visualization of  elements is difficult. Thus, a sensitivity analysis can focus on different aspect:

Overall sensitivity: here we take the maximal value of the elements in , i.e.,




- Sensitivity with respect to selected input-output pairs. Here, one or two inputs and one output are selected and the resulting graph is plotted.
- Adaptive sensitivity analysis: in this approach, the entire functional envelope is covered with test points . At each point, the sensitivity is determined. Given a smoothness assumption on the neural networks (i.e., the learned function does not exhibit any abrupt changes), we can say that within a certain area around , the sensitivity does not change considerably. Thus, we can guarantee that the NN's behavior is correct in this area (given that  produces the right result). If the sensitivity at  is low, this area can be considerably large, thus reducing the number of required test-data. For further details on this approach see [24, 26].

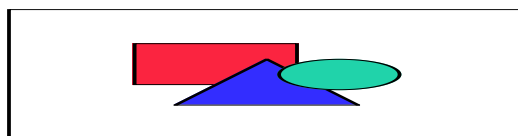
**Example** For illustration, let us consider a two-dimensional function in  as shown in Figure 17. In this function, two smooth areas with low gradients are separated by a somewhat sharp (but still differentiable) transition. One can imagine that such a function<sup>1</sup> could show up on a transition from sub-sonic into supersonic flight.



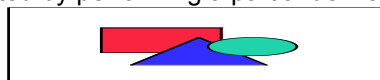
**Figure 17: Graph of a two-dimensional function (exact representation).**

We have used a simple feed-forward network (2 input node, 8 hidden node, 1 output node) to learn this function from a set of equally-spaced training data (656 data points). For this training, a standard back-propagation algorithm has been used.

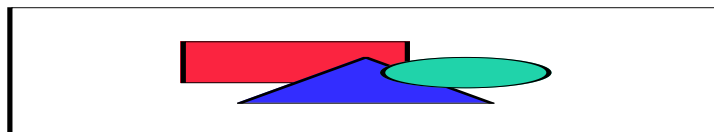
In order to analyze the sensitivity of the output with respect to the inputs, we have to calculate the Jacobian . For a simple feed-forward network, this calculation is straight forward. The output function of the NN is given by the vector equation:



where  $n_h$  is the number of hidden nodes,  $n_i$  the number of input nodes, and  $b_h$  the biases on the hidden nodes. The elements of the Jacobian now can be calculated by performing a partial derivative and using



the chain rule. The derivative of the hyperbolic tangent is  $\frac{1}{4} (1 - \tanh^2(x))$ . We now obtain:



**Example** When calculating the Jacobian of the network representation of our function from Figure 17, we obtain the following graph as shown in Figure 18. This mesh displays the sensitivity of the output with

respect to the input  $x$  on our operational envelope. It is easy to see that the sensitivity is low in the smooth areas of the envelope and, (as expected) high in the transition area. Therefore, a close analysis of this region is strongly advised.

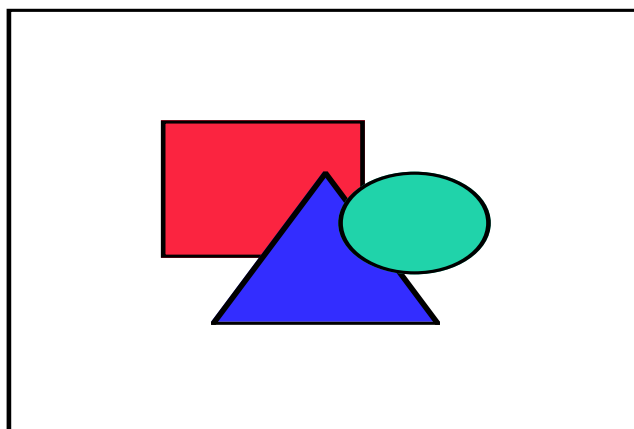
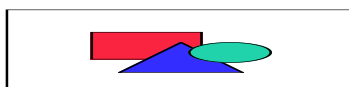


Figure 18: Graph of the Jacobian of the NN representation with respect to  $x$ .

## 13.7. Condition Numbers

For the analysis of the sensitivity of the neural network with respect to the training, it is often helpful to look at the quadratic form of the error function (which is minimized during training). Please note, that for the sensitivity analysis in the last section, we considered the influence of small perturbations in the input on the output. Here, we consider, how small changes in the weights affect the error function. Let  $\mathbf{w}$  be a quadratic representation (or approximation) of the error function  $E(\mathbf{w})$  in the neighborhood of the (local)

minimum,  $\mathbf{w}_0$ .  $\mathbf{w}_0$  is defined as



where  $\nabla f(\mathbf{x})$  is the gradient, and  $\mathbf{H}(\mathbf{x})$  is the Hessian matrix. It is a positive definite matrix. In the vicinity of  $\mathbf{x}^*$ , the behavior of  $f(\mathbf{x})$  (and thus also of  $\nabla f(\mathbf{x})$ ) is determined by the eigensystem of  $\mathbf{H}(\mathbf{x}^*)$ . This directly follows from the fact that at a stationary point (and a minimum is such one), the gradient must be zero. With

$$\nabla f(\mathbf{x}) \approx \mathbf{H}(\mathbf{x}^*) (\mathbf{x} - \mathbf{x}^*)$$

and setting this equation to zero, we obtain

$$\mathbf{H}(\mathbf{x}^*) (\mathbf{x} - \mathbf{x}^*) = \mathbf{0}$$

In the vicinity of  $\mathbf{x}^*$ , e.g.,  $\mathbf{x} = \mathbf{x}^* + \mathbf{v}$  we now obtain

$$\mathbf{H}(\mathbf{x}^*) \mathbf{v} = \mathbf{0}$$

and with Equation 7 this simplifies to

$$\mathbf{H}(\mathbf{x}^*) \mathbf{v} = \mathbf{0}$$

Using the definition of eigenvectors and eigenvalues of  $\mathbf{H}(\mathbf{x}^*)$ , namely for eigenvalues  $\lambda_i$  and eigenvectors  $\mathbf{v}_i$

$$\mathbf{H}(\mathbf{x}^*) \mathbf{v}_i = \lambda_i \mathbf{v}_i$$

$$\mathbf{H}(\mathbf{x}^*) \mathbf{v}_i = \lambda_i \mathbf{v}_i$$

we get

$$\mathbf{H}(\mathbf{x}^*) \mathbf{v}_i = \lambda_i \mathbf{v}_i$$

When we try to visualize this for two dimensions, we get an ellipsis around  $\mathbf{x}^*$  as shown in Figure 19. The large major axis is along the eigenvector  $\mathbf{v}_1$ , the other axis along  $\mathbf{v}_2$ . The length of the axes is determined by the absolute value of the eigenvalues  $\lambda_i$ .



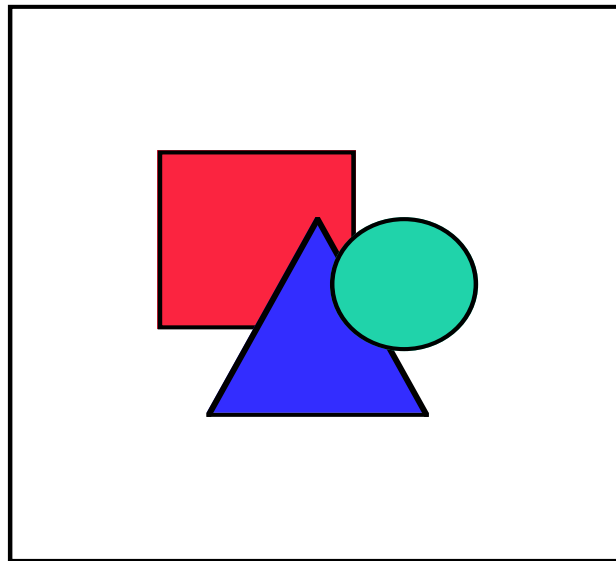


Figure 19A

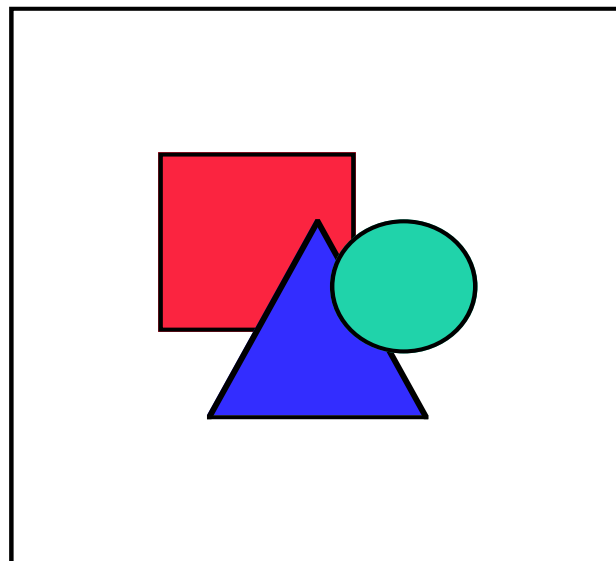


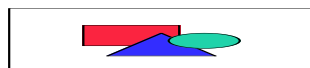
Figure 19B

**Figure 19: Contours of a well conditioned (A) and bad conditioned problem. The minimum is at the center of the ellipses, the thick arrows mark the semi-axes of the ellipses.**

With this observation, we now can figure out what the definition of the condition number actually means. The condition number is defined as the ratio of the largest eigenvalue and the smallest one, i.e.,



where



are the eigenvalues in descending order. If all eigenvalues are of similar magnitude, the condition number goes toward 1. In the 2-dimensional case,

this means the contours of the function in vicinity of the minimum are circles (or very round ellipses). This situation, as shown in Figure 19A, behaves very well. For any search algorithm, the search direction and an appropriate step size can easily be found. Thus, good convergence can be expected.

If, however, the condition number is large, we can expect trouble. In that case, the contours are very small, valley-like ellipses as shown in Figure 19B. Because the steepness substantially changes in the different directions, care must be taken not to choose a step-length which is too large. Whereas a long step along the "valley" is in order (i.e., it decreases the function value), a step of the same length in the orthogonal direction leads into the opposite "wall", resulting in oscillation or divergence of the algorithm. Although the condition number gives a good indication of the expected numerical behavior of the problem, the actual computational costs of calculating this number are considerable. There are two major time-consuming steps involved: calculation of the Hessian, and the calculation of the eigenvalues. For a

Neural Network, the Hessian is of size  $n \times n$  where  $n$  is the number of weights in the NN. Thus its size can be quite substantial. [24] describes an elegant way to calculate the Hessian for a feed-forward network, using a back-propagation style of algorithm.

Nevertheless, in practice, it might be advisable to look at approximations of the Hessian matrix. A straightforward approximation can be obtained, if the Hessian is not calculated directly, but rather in the form

$$H \approx L D L^T$$

where  $L$  is a triangular matrix, and  $D$  is a diagonal one. Because the Hessian is positive definite, this

factorization always exist. Then, we can take the elements  $d_i$  of the diagonal matrix in the same way as described above to obtain an approximation of the condition number:

$$\kappa \approx \frac{d_{\max}}{d_{\min}}$$

This approximation is a lower bound of the condition number. Using the  $L D L^T$  form of the Hessian can have substantial advantages in training algorithms which use second derivatives (for details see [27]). According to [26], there are cases where some small additional computational overhead should be used to obtain better approximations of the condition number. One of these methods is described in the following. Let us define the vectors

$$v_i = \frac{1}{\sqrt{d_i}} L^T e_i$$

and

$$w_i = \frac{1}{\sqrt{d_i}} L e_i$$

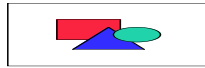
where  $i_{\max}$  is the index of the maximal diagonal element and  $i_{\min}$  that of the minimal.

$$v_{i_{\max}}^T H v_{i_{\max}}$$

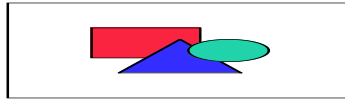
. It can be shown that

$$v_{i_{\max}}^T H v_{i_{\max}} = d_{i_{\max}}$$

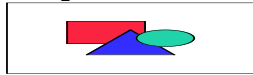
and



and consequently



This gives us the lower bound on the condition number. If we normalize these vectors, i.e.,







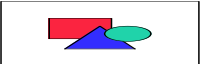
, it can be shown that , and similarly . For details and an example see [26].

## 13.8. Analysis of the Training Algorithm




In this section, we will discuss issues on how (and how fast) a training algorithm will proceed from a given starting point to a minimum, and how we can stop the training algorithm when it came "sufficiently close", that is, if it does.


In general, NN training algorithms are iterative numerical optimization algorithms which try to find a (global) minimum. The function which needs to be minimized is the error function. NN training algorithms

usually have the following iterative structure. From a starting value , the algorithm proceeds using the following steps:

1. Determination of search direction  to go from ,
2. Determination of the length of a search step, ,
3. Go toward that direction, i.e., .
4. If we are *done*, then *stop*, else *goto* 1

So far, the training algorithm looks pretty straight-forward. However, there are a number of problems that can lead the search astray or grind it to a halt (see the following subsection). Always remember, that for multivariate functions (i.e., a function with more than one input), all these algorithms only converge (if they do) to a local minimum. There is no guarantee that they will reach a global minimum.

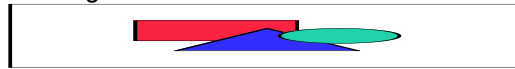
As will be seen in the following, the determination of search direction  and step length  has a crucial effect on the algorithm's behavior and performance. Therefore, a large number of different algorithms and variants have been developed over time. They differ not only how both values are calculated, but also in what kind of information is available about the error function  (e.g., availability of 2nd derivative). In the following subsection, we will discuss issues concerning the *progress* of the training algorithm.

The training algorithm stops in Step 4 when  is "sufficiently close" to the desired minimum. In Section 12.8.2, we will focus on details on the stopping criteria.

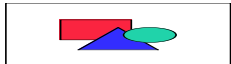
### 13.8.1. Progress of the Training

Despite the large variety of different training algorithms and their variants, it is far from trivial to select an algorithm which performs optimally (or even well) for a given problem. The following example will illustrate this.

**Example** Let us consider the following two-dimensional function



This function is known as *Rosenbrock's banana function*. This function (shown in Figure 20) has a unique

minimum at point . Figure 21 shows a logarithmic contour plot of this function. In this plot, lines connect points of equal function values - similar to elevation lines on a hiking map.

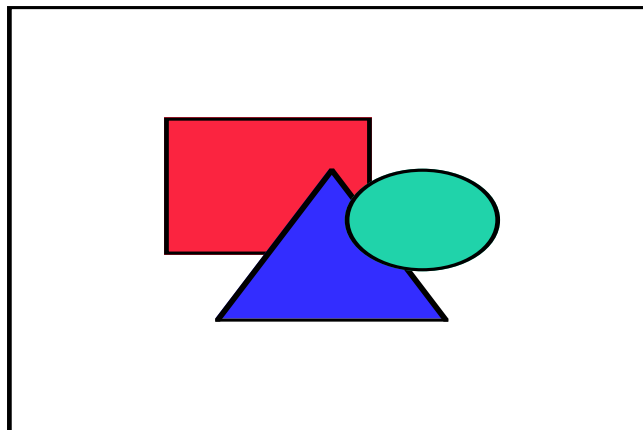


Figure 20: 3D-plot for Rosenbrock's banana:

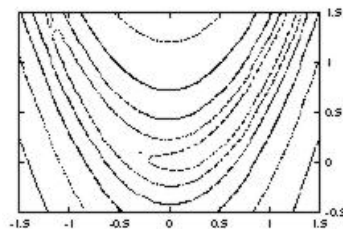
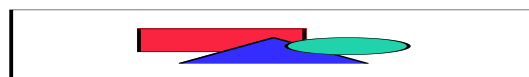


Figure 21: Contour-plot for Rosenbrock's function (logarithmic z-axis). The starting point  and the global minimum  are marked.

We now use different algorithms to find the (global) minimum of this function. For all algorithms, the

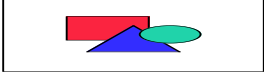

starting-point is . Then, the individual training steps are overlaid the contour plot. This experiment is described in detail in [26].

Figure 22 shows the behavior of three different search algorithms. The left figure depicts a good, albeit not optimal behavior. After starting, the algorithm moves along the "valley" to reach the global minimum after a small number of steps. The graph in the middle shows, how an algorithm can get stuck. After a number of reasonable steps, the step-length becomes smaller and smaller. Thus the search process gets slower until it grinds to a halt, far away from the minimum. Finally, in the picture on the right hand side, the algorithm suddenly diverges. It produces iterations with large step-sizes, but the search direction does not go toward the minimum. In that case, an arbitrary result can occur. It is surprising, how the performance for such a smooth, relatively simple (but nonlinear) two-dimensional varies. In general, situations where the function which has to be minimized exhibits narrow "valleys" can be a source of convergence problems.

In our NN training case, where we usually have a quadratic error function which needs to be minimized, these valleys are of elliptic shape (cf. Figure 19). There, narrow valleys correspond to an ill-conditioned

problem. The ellipsis around the minimum  is characterized by the minor and major semi-axes. Their ratio defines the shape of the ellipsis; therefore, a narrow one has a large major and short minor semi-axis. On the other hand, these semi-axes directly correspond to the eigenvectors/values of the Hessian. Thus, a narrow ellipsis corresponds to a large ratio between the eigenvalues, and thus a badly conditioned system (see Section 12.7 for details).

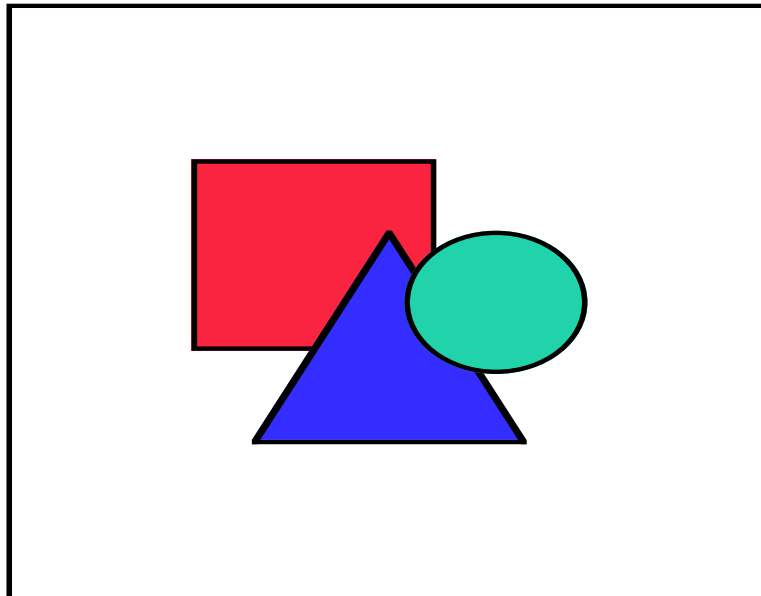


Figure 22A

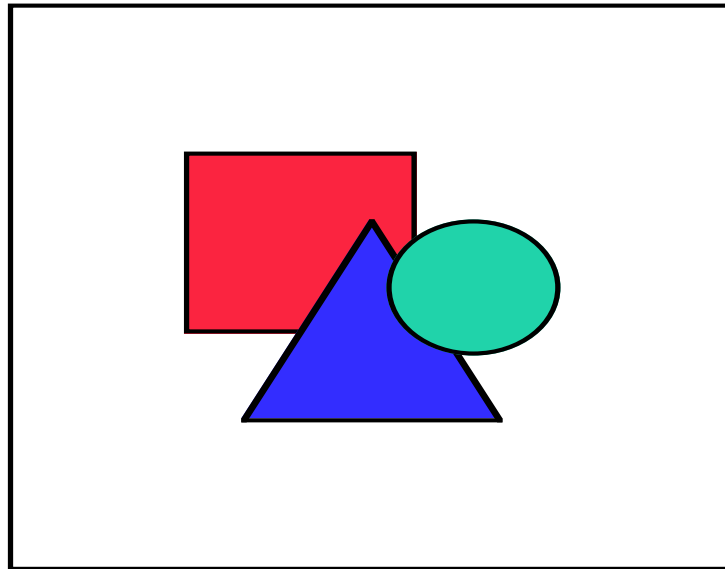


Figure 22B

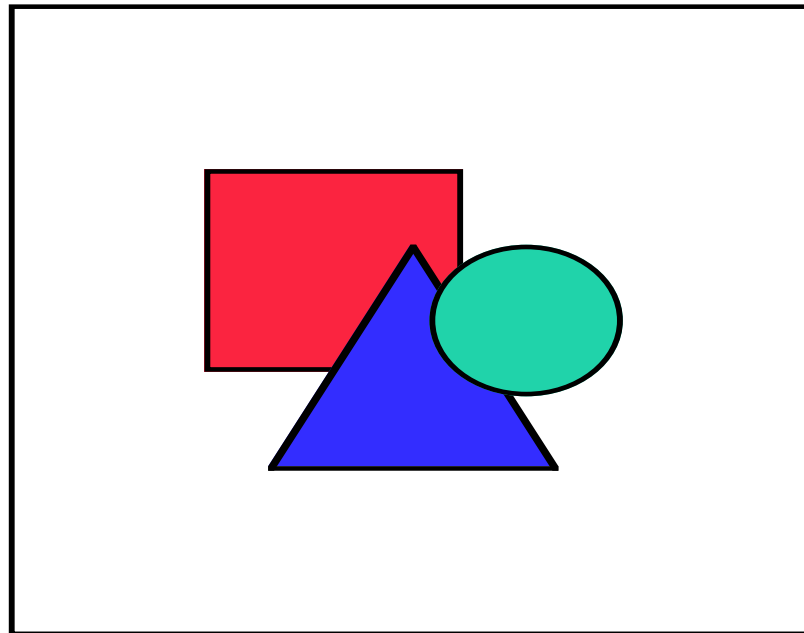


Figure 22-C

Figure 22: Behavior of different search algorithms on Rosenbrock's function.

### 13.8.2. Stopping (the Training)

*“Begin at the beginning”, the king said gravely, “and go on till you come to the end; then stop.” Lewis Carroll - Alice in Wonderland*



In this section, we discuss, “what it means when we stop”:

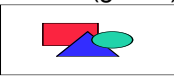
- Have we solved the problem?

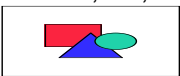


- Have we ground to a halt? (i.e., no more progress)
- Have we exhausted resources (time, memory, patience (yes!))?
- Have we solved the right problem?

**Note:** In this section, we will not discuss the issues of stopping the training with respect to number of data vectors in the training set (and issues like over-training).

Stopping, of course, only applies to the iterative training algorithm. An iterative training algorithm starts

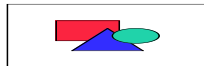
from a point . Then, it iteratively tries to reach a (global) minimum . A necessary condition for a

minimum is that the gradient there is 0, i.e., . However, due to finite-precision machine



arithmetic, we only can obtain . Furthermore, it needs to be checked, that the point  is *not* a saddle point. Here, we can benefit from the theorem that for a minimum , the Hessian is *positive definite* (see Section 12.4). For a one-dimensional function, this boils down to the well-known: the second derivative at this point must be larger than zero. However, in general, calculation of the Hessian is expensive. In the following, we discuss several, well-known stopping criteria with respect to their efficiency and effect on scaling. It must be said that there is no "one-size-fits-all" stopping criterion.

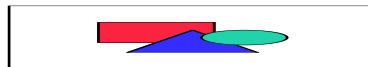
### 13.8.2.1. Absolute Stopping Criteria

The straight-forward stopping criterion





usually does not work well, since it is not stable with respect to scaling (see also Example 12.5.3). In

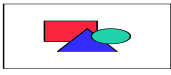
order to overcome the scaling problem for  (but not for ) , the following condition can be used [28]

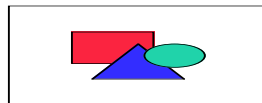


### 13.8.2.2. Relative Gradient Stopping Criterion

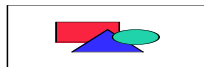
Another possibility to overcome effects of scaling is to use a *relative* measure of the gradient, defined as

the ratio of the relative rate of change in  over the relative rate of change in . Thus, we obtain for the

individual components of this relative gradient :



Then, the stopping criterion is



This stopping criterion is not affected by scaling. However, when the minimum, or the function value close to the minimum approaches 0, round-off errors can occur.

### 13.8.2.3. Test for Convergence or Stalling

Instead of testing whether we are close to a minimum (i.e., see if the gradient gets sufficiently small), we can monitor the *progress* of the algorithm. Because most optimization algorithms (but not standard backprop) reduce the length of each step in the vicinity of a minimum, we can stop, whenever the relative change of the  $\|x\|$  value between iterations gets sufficiently small. This means,

$$\frac{\|x^{(k)} - x^{(k-1)}\|}{\|x^{(k)}\|} < \epsilon$$

where  $x^{(k)}$  is the x-value at the  $k$ -th iteration. Norm is just a normalizing operator to overcome scaling problems.

As a rule of thumb, the value of  $\epsilon$  can be set as follows: if the calculation requires  $N$  valid decimal digits, the tolerance should roughly be set to  $10^{-N}$ .


This stopping criterion does terminate the algorithm not only when a minimum is reached. Also in situations, where the algorithm gets "stuck" (e.g., in shallow places), this stopping criterion fires. It is therefore important that, after the algorithm stops, the current values of  $x$  are checked with respect to the minimum (e.g., by using one of the above criteria).

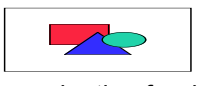
### 13.8.2.4. Stopping after N iterations


In most practical applications, it is meaningful to set an upper limit on the number of iterations. This limit circumvents the problem that the algorithm can run for an arbitrary amount of time. For a system with timing constraints (e.g., for on-line training), such a limit is an absolute must.

However, two things need to be taken into account:

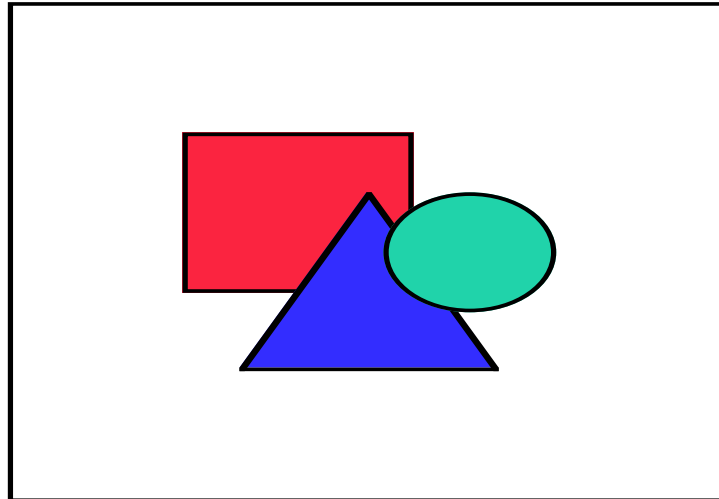
- If the algorithm stops after  $N$  iterations, it needs to be checked to determine if the algorithm actually reached a minimum, or if it got stuck.
- If the actual run-time is to be limited by this means, care has to be taken that the run-time for each iteration is exactly the same. If numerical subroutines (e.g., matrix inverse), iterative

subroutines (e.g., , or sparse matrices are used, this condition might be violated.

**Example** Usually, the runtime of a numerical library function (e.g., ) is considered to be small and constant. However, due to the built-in algorithms, function evaluation for different parameter values can take different amounts of time. These effects, as shown in Figure 23 can be substantially large

(here, approximately ) . In particular in time-critical applications, these effects have to be analyzed (and tested) carefully.

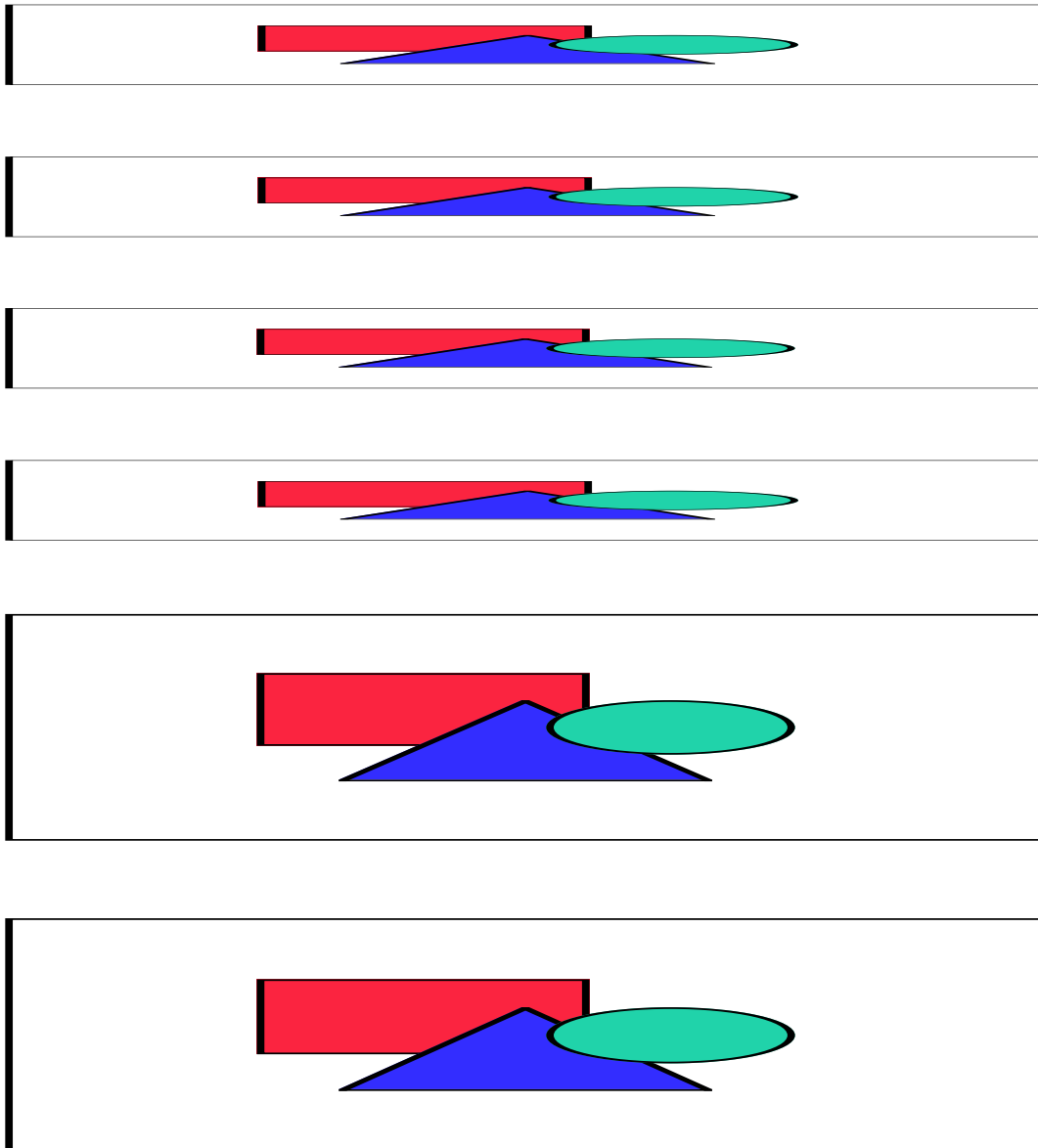


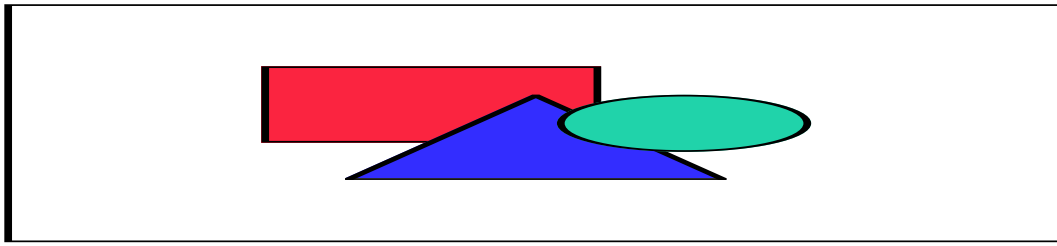


**Figure 23: Run-time of library sin-function over a wide range of input values (log-scale). Run-time in seconds for 50'000 evaluations (on a Linux notebook).**

## 14. APPENDIX I: BASICS

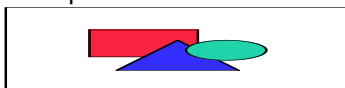
In the following, we assume that  $f$  is a function from  $X$  into  $Y$ . Extreme values, like minimums and maximums can be defined as follows:








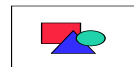
# 15. APPENDIX J: QUADRATIC FUNCTIONS AND QUADRATIC FORMS

An arbitrary non-linear function can be expressed as a Taylor series. Because in our context, we only consider relatively smooth functions, it is sufficient to consider only the first few terms. In particular, the approximation to quadratic functions is of importance. Let

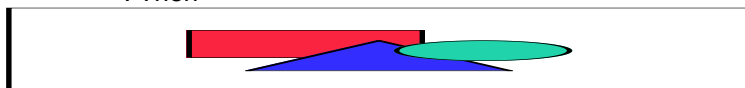



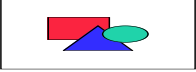
be such a quadratic function with , and the Hessian . Then .

Let us now consider the behavior of  around some point , namely  for , and

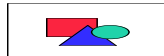


. Then



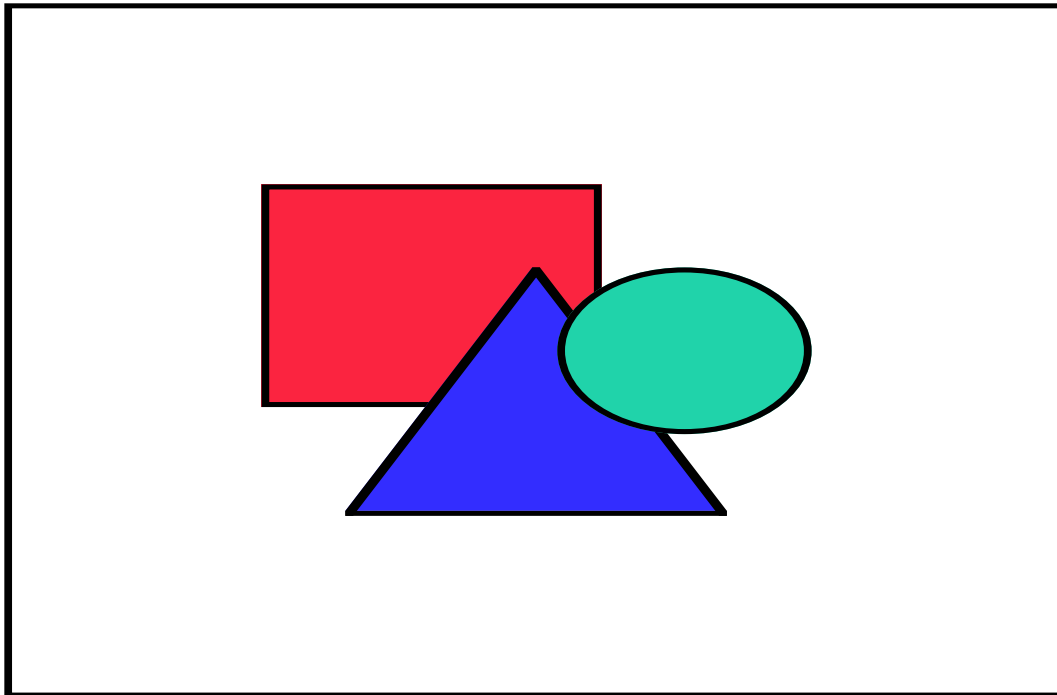
For a stationary point  (which is a pre-requisite for a minimum) we need to have: . On

the other hand,  (by differentiation of the definition of ). Thus we have









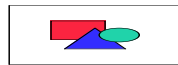
which is a system of *linear equations*.


Therefore, non-linear optimization problems can be handled, in the vicinity of a stationary point as a set of linear equations.

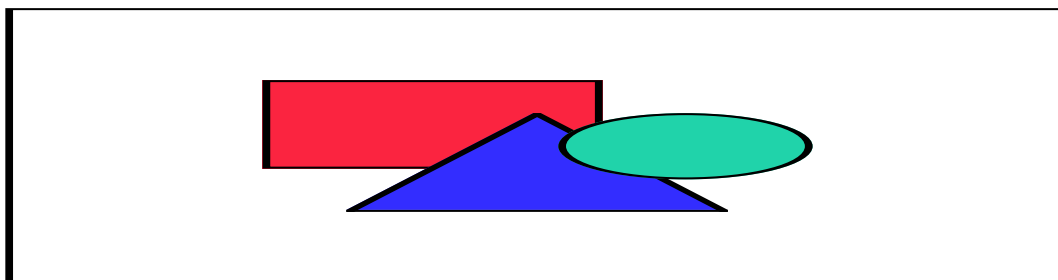
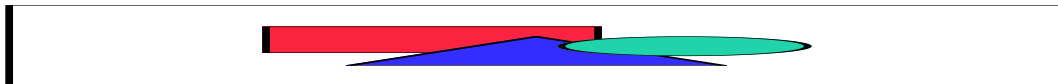
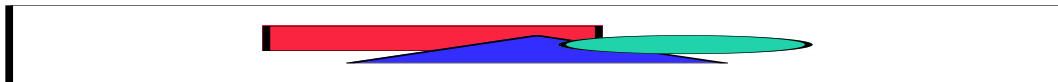
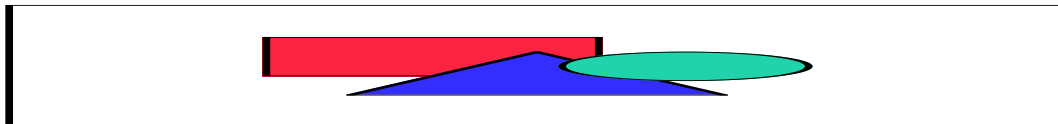



## 16. APPENDIX K: EIGENVECTORS AND EIGENVALUES

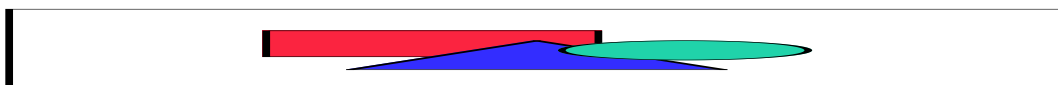
For a  matrix , a set of vectors  is called *eigenvectors* of , and  are called *eigenvalues* if for all 



The eigenvalues correspond to the solutions of the characteristic equation for the matrix ,



Corollary:  for a positive definite matrix





## 17. REFERENCES

- <sup>1</sup> Jo'se Principe, Neil Euliano and W., Curt Lefebvre, *Neural and Adaptive Systems, Fundamentals through Simulations*, John Wiley, 2000.
- <sup>2</sup> Russell D. Reed and Robert J. Marks, II, *Neural Smithing*, Massachusetts Institute of Technology, 1999. p. 2
- <sup>3</sup> *Definition for the Verification And Validation of Neural Networks For Aerospace Applications*, February 19, 2002, Dale MacKall, Dryden Space Flight Center
- <sup>4</sup> Reed and Marks, *Neural Smithing*, P. 7
- <sup>5</sup> S. Y. Kung, *Digital Neural Networks*, PTR Prentice Hall 1993. p. 30
- <sup>6</sup> Interview with Chuck Jorgensen, March, 2002
- <sup>7</sup> S. Y. Kung, *Digital Neural Networks*, p. 27
- <sup>8</sup> Digital Neural Networks S. Y. Kung PTR Prentice Hall 1993. pp 85-86
- <sup>9</sup> *NASA Procedures and Guidelines NPG: 2820.DRAFT, NASA Software Guidelines and Requirements as of 3/19/01* (Responsible Office: Code AE/Office of the Chief Engineer), NASA Ames Research Center, Moffett Field, California, USA
- <sup>10</sup> IEEE Standards 12207.0, 12207.1, 12207.2 located at the following web address (URL):  
[http://ieeexplore.ieee.org/search97/s97is.vts?Action=FilterSearch&SearchPage=VSearch.htm&ResultTemplate=adv\\_crst.hts&Filter=adv\\_sch.hts&ViewTemplate=lpdocview.hts&query1=12207&scope1=&op1=and&query2=&scope2=&op2=and&query3=&scope3=&collection=jour&collection=conf&collection=stds&collection=pprint&py1=&py2=&SortField=pyr&SortOrder=desc&ResultCount=15](http://ieeexplore.ieee.org/search97/s97is.vts?Action=FilterSearch&SearchPage=VSearch.htm&ResultTemplate=adv_crst.hts&Filter=adv_sch.hts&ViewTemplate=lpdocview.hts&query1=12207&scope1=&op1=and&query2=&scope2=&op2=and&query3=&scope3=&collection=jour&collection=conf&collection=stds&collection=pprint&py1=&py2=&SortField=pyr&SortOrder=desc&ResultCount=15)
- <sup>11</sup> Reed and Marks, *Neural Smithing* p. 170-172
- <sup>12</sup> Reed and Marks, *Neural Smithing* p. 1
- <sup>13</sup> Reed and Marks, *Neural and Adaptive Systems* p. 570
- <sup>14</sup> Reed and Marks, *Neural Smithing*, p 317
- <sup>15</sup> Jose Principe, et al, *Neural and Adaptive Systems, Fundamentals through Simulations*
- <sup>16</sup> Reed and Marks, *Neural Smithing* p.71-72
- <sup>17</sup> Microsoft Corporation. "Windows 2000 Server Resource Kit Online Books", MSDN Library Glossary. 1985-2000.
- <sup>18</sup> Whatis?com [http://whatis.techtarget.com/definition/0.,sid9\\_gci284015,00.html](http://whatis.techtarget.com/definition/0.,sid9_gci284015,00.html)
- <sup>19</sup> Reed and Marks, *Neural Smithing*, p. 127
- <sup>20</sup> John Kaneshige and Karen Gundy-Burlet, *Integrated Neural Flight and Propulsion Control System* American Institute of Aeronautics and Astronautics, AIAA-2001-4386



- 
- <sup>21</sup> Snns: Stuttgart neural network simulator. URL: <http://www-ra.informatik.uni-tuebingen.de/SNNS/>, 2002.
- <sup>22</sup> M. C. O. M. I. B. S. Report). Report on project management in nasa by the mars climate orbiter mishap investigation board. URL: <http://www.nasa.gov/newsinfo/marsreports.html>, 2000.
- <sup>23</sup> W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge Univ. Press, Cambridge, UK, 2nd. edition, 1992.
- <sup>24</sup> D. Soloway. Improved convergence for output scaling of a feedforward network with linear output nodes. In *1993 IEEE International Conference On Neural Networks*. IEEE Computer Society Press, 1993.
- <sup>25</sup> C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon-Press, Oxford, 1995.
- <sup>26</sup> J. Schumann. Vericonn: Verification of controllers based on adaptive neural networks -- white paper-- . Technical report, NASA Ames, Automated Software Engineering, 2001.
- <sup>27</sup> P. Gill, W. Murray, and M. Wright. *Practical Optimization*. Academic Press, 1981.
- <sup>28</sup> J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, volume 16 of *Classics in Applied Mathematics*. SIAM, 1996.